NUS CS-CS5562: Trustworthy Machine Learning August 25, 2023

Assignment 1

Lecturer: Reza Shokri

The assignment contains the following six parts. Task 1 to 4 will be graded.

1. **Warm up:** Implement the PGD and FGSM algorithms to attack a pre-trained NN model to get familiar with the pipeline.
2. **Task 1: Adversarial attacks on self-driving cars**
3. **Task 2: Obfuscating gradients for robustness**
4. **Task 3: Design your adaptive attack**
5. **Task 4: Build a defense anticipating the adaptive attack**
6. **Task 5 (Optional): Certify the robustness of the model**

You are asked to complete the code in **designated** places in the Jupyter Notebooks. You are **not** allowed to change any other parts of the notebook unless you are told to do so. You also need to write a short report in the Notebook for Task 3 and Task 4.

- You are required to submit a Zip file (`.zip`) containing the Jupyter notebooks only.

Setting up the environment

All the Jupyter notebooks can be run on Google Colab [*] or your local machines.

Google Colab

To use Google Colab, you need to first upload the entire assignment folder to your Google Drive. After launching individual notebooks, you can choose to use GPU as your runtime. The instructions to mount your Google Drive is included in the notebook. You should be able to run all the code without installing any packages in Colab.

Local Machine

Although we use mostly standard libraries in the assignment, there is a chance your local environment is missing one or two packages or have a version mismatch. We strongly recommend using Anaconda to install and manage the packages used in this assignment. Once Anaconda is installed, go to the assignment's root directory, where you will see `environment.yaml`. Install all the required packages by typing this command into the terminal

```
conda env create --file environment.yml
```

You should have an environment called `cs5562` with all packages needed to run the code. Switch to this environment by

```
conda activate cs5562
```

If installing all packages takes too long on your machine, you can specify this before creating the environment:

```
conda config --set channel_priority flexible
```

[*] https://colab.research.google.com/

Warm up: Adversarial examples

In this task, you will get familiar with the code and attack pipeline through implementing PGD and FGSM attacks.

Implementation of standard attack algorithms

In the lecture, you have learned about the FGSM attack algorithm (Goodfellow et al. [2014]) and the PGD attack algorithm (Madry et al. [2017]). Now, let's implement them. The target model is a ResNet50 (He et al. [2016]) model pre-trained on ImageNet (Deng et al. [2009]). We consider the commonly-used CrossEntropyLoss as the loss function, and ℓ_∞ ball as our default perturbation set.

1. Implement the `attack` function for the `FSG_attack` class in the Warm-ups notebook. You are given the target model `target_model`, the perturbation budget `epsilon` and also a clean image `test_image` along with its true label `y_true`. The function should output a perturbation δ within the l_∞-ball of size ϵ (i.e., $||\delta||_\infty \leq \epsilon$).

   You need to implement the untargeted attack (where the adversary wants to "fool" the model to output a wrong label on adversarial examples) and also the targeted attack (where the adversary wants to "fool" the model to output target label on adversarial examples). The `y_target` is the target label. The flag `is_targeted` indicates which attacks should be performed.

2. Implement the `attack` function for the `PGD_attack` class in `attack.py`. The parameters are similar to the previous attack. In addition, you are also given a parameter `step` indicating how many gradient ascent steps need to be performed, and the learning rate `learning_rate`. Similarly, you need to implement the untargeted attack and the targeted attack.

Task 1: Adversarial attacks on self-driving cars

When creating a self-driving car, one important capability is negotiating bends and turns based on the camera feed of the road ahead. Machine learning is a candidate to implement such a functionality, and in 2016, Udacity released a challenge to use machine learning to perform steering angle prediction for self-driving cars. The dataset consists of 480 x 640 RGB images and the corresponding steering angles of the car at the moment the frame was captured. The images are continuous frames of a video feed from a car driving.

We will be using a particular community model released by Udacity named cg23 [†] in the assignment. The 480 x 640 images are cropped and transformed into 128 x 128 RGB images. The input for the model is a 128 x 128 RGB image of the road in front of the car, and the output is a real number representing the steering angle (positive for right, negative for left).

$$F : \{0, \dots, 255\}^{128\times128\times3} \to \mathbb{R}$$

In this task, we will perform an attack (JSMA) on a given trained cg23 model.

Jacobian-based Saliency Map Attack (JSMA) The JSMA scheme was proposed by Papernot et al. [2016]. It is also a gradient-based attack which computes the gradient of the loss with each class label with respect to all components of the input. The gradient is in the form of a matrix, and is referred to as the Jacobian matrix. Intuitively, the Jacobian matrix measures the sensitivity of output label with respect to each entry of the input.

In this exercise, we will implement a simpler algorithm that utilizes the Jacobian instead of the saliency map to select the to-be-perturbed pixels. We have also provided the code snippet that computes the Jacobian and reshapes it to the shape of an input image.

Objective 1

We play the role of an attacker with the knowledge that there is a self-driving car using the trained cg23 model for its steering angle prediction. However, our attack is restricted to the physical world, i.e. we cannot directly change the camera feed before it is given to the model. One way to do this is to perform an attack constrained on a small section of the input that **we can physically change**. For instance, Fig 1 is the camera feed the model might receive. The two signboards in the image (Fig 2) are what we can physically change, such that when the self-driving car using the cg23 model passes by the same signboard, the steering angle prediction model will mispredict and cause the car to swerve off the road. **NOTE**: The model itself is not fully accurate in its prediction. Hence in this assignment, we care only about creating perturbations such that the model deviates significantly from its **original prediction**, in any direction.

[†] https://github.com/udacity/self-driving-car/tree/master/steering-models/community-models/cg23

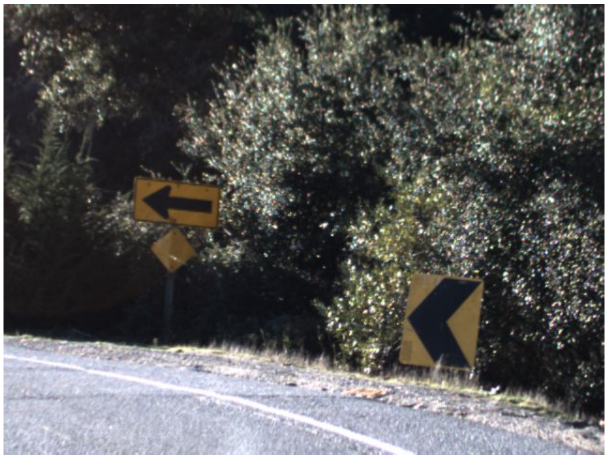


Figure 1: An example of the camera-feed image from a self-driving car while driving.

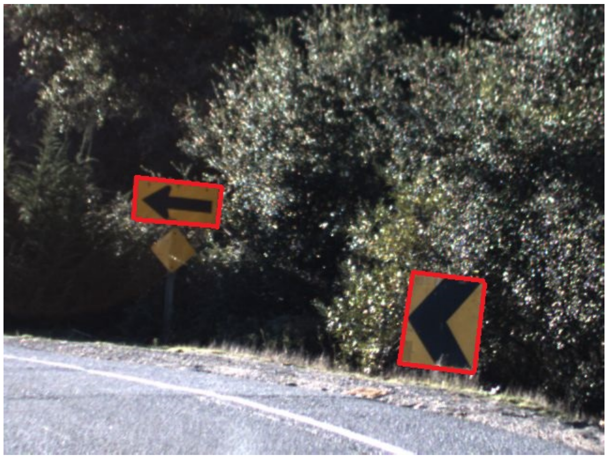


Figure 2: Figure 1 with the area of allowed perturbation highlighted.

Implementation details

Complete the algorithm in `JSMARegressionAttack.attack_batch` in the Task 1 notebook. Before that, download the pre-trained model from the link and save the extracted directory under the `JSMA/` directory. Make sure you have `JSMA/models/sdc-50epochs-shuffled`. You should change only **one** distinct pixel to perturb for each input image in each iteration. Refer to the comments in the TODO block for hints.

Objective 2

After completing Part 1, we have successfully created individual constrained perturbations for single frames. However, note that the individual perturbations in each frame of the video are inconsistent, meaning if played back as a video, the perturbations change

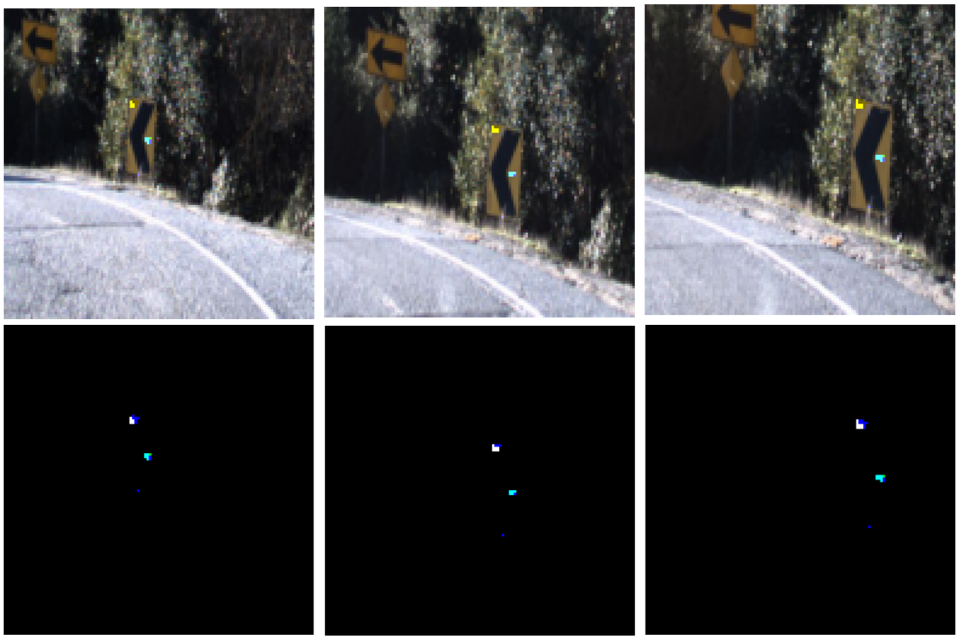


Figure 3: Top row shows a perturbation that is consistent across frames in a video. Bottom row shows that perturbations without the original image.

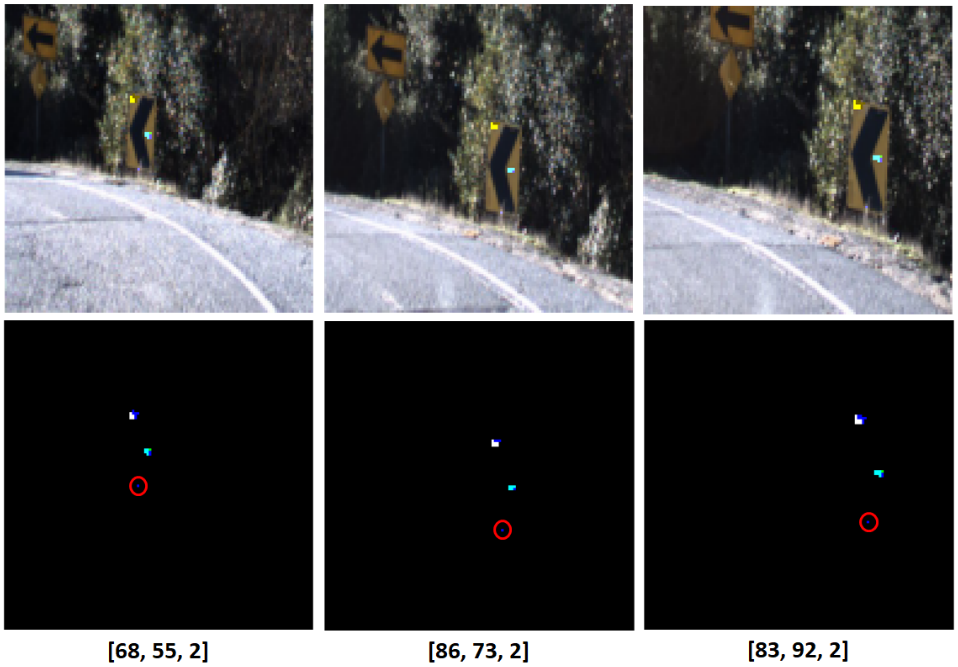


Figure 4: Same as Figure 3, with particular vertices, circled and labeled. Note that the last element is 2 because the circled vertices are blue.

from frame to frame.

We could naively use the perturbations from one of the frames (say, the last frame) to physically change the signboard, but the robustness of the attack would not be as high since we only considered a single frame for the attack. Recall that in the context of a self-driving car, the car will be receiving steering predictions every few frames. If the model only mispredicts significantly for a single frame, the car likely would not swerve off the road as intended.

The goal of this part of the assignment is, therefore, to create a single **consistent** perturbation, which will be applied across the sequence of frames of a video slice and causes the model to mispredict significantly on all frames of the attack target.

For this assignment, this will be achieved with the help of a **pixel mapping** function, which accepts a sequence of frames, and returns a list of pixel (vertice) strings. Each vertice string is a list of equivalent vertices in the sequence of frames. You should change only **one** distinct vertice string to perturb for each input sequence in each iteration.

Implementation details

Complete the algorithm in `JSMARegressionAttack.attack_pixelmap` in the Task 1 notebook. Refer to the comments in the TODO block for hints.

Task 2: Build defense based on obfuscated gradients

In this task, you take the role of the defender, and you need to build a robust model against adversarial examples, i.e., it can classify adversarial images correctly with little performance loss on non-adversarial (clean) images.

In the lecture, you have learned that gradient masking is type of defense. Here, we focus on one defense method from this category (proposed in Xie et al. [2017]) which utilizes randomization at test time to mitigate adversarial effects. More specifically, the defender uses two randomization operations at the test time: random resizing, which resizes the input images to a random size, and random padding, which pads zeros around the input images randomly. Figure 5 shows the framework of the defense algorithm. The algorithm adds a random resizing layer and a random padding layer to the beginning of the classification networks. There is no re-training or fine-tuning needed, which makes the proposed method very easy to implement. In this way, the adversary cannot get the accurate gradient, and we hope that the attacks will not be effective anymore.

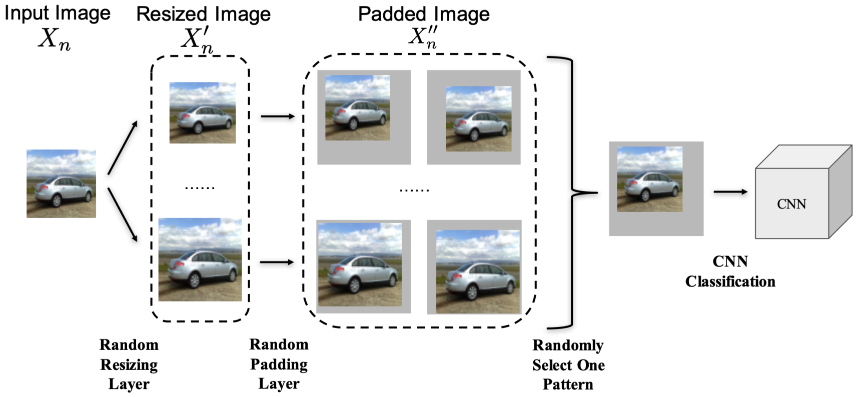


Figure 5: The pipeline of our randomization-based defense mechanism (see Xie et al. [2017]). The input image X first goes through the random resizing layer with a random scale applied. The random padding layer pads the resized image X'_n in a random manner. The resulting padded image X''_n is used for classification.

Implementation details

Implement `randomization_input` in the Task 2 notebook. You are given the input image `test_image` and the `resize_bound` indicating the upper bound of the resize operation. The function outputs the randomized test images.

Task 3: Break the defense

In the previous task (Task 2), you have managed to build a defense model. It is simple and effective. However, the adversary who knows the defender's strategy can design a more effective attack algorithm to bypass the defense in practice. The defense model, which is robust against the existing attacks, might fail to be robust against adaptive attacks. In this task, you need to design your adaptive attack to break the defense model (implemented in Task 2).

Implementation details:

You need to design your own attack algorithm and implement it in the `attack` function of the `adaptive_attack` class in the Task 3 notebook. The parameters are the same as those in previous attack algorithms. You need to implement both the targeted attack and the untargeted attack.

Report:

You need to include the design of your attack algorithm in the report section of the notebook. You can also inform us about any difficulties you have faced and how you solve them.

Task 4: Adversarial Training

> Know yourself and know your enemy, and you will never be defeated (from Sun Tzu's the Art of War).

The defense based on obfuscated gradients fails to be robust against adaptive attacks. This demonstrates that when designing a defense model, we must consider adaptive attacks. Hence, the objective of training a robust model should be formalized as follows:

$$\min_{\theta} \frac{1}{|S|} \sum_{(x,y)\in S} \max_{\delta\in\Delta} \ell(h_\theta(x+\delta), y), \tag{1}$$

where S is the training dataset and Δ is the l_∞-ball of size ϵ. As with traditional training, the way we would solve this optimization problem in practice is to use stochastic gradient descent over θ. That is, we would repeatedly choose a minibatch $B \subseteq S$, and update θ according to its gradient.

$$\theta := \theta - \frac{\alpha}{|B|} \sum_{(x,y)\in B} \nabla_\theta \max_{\delta\in\Delta} \ell(h_\theta(x+\delta), y). \tag{2}$$

Based on DanskinŹs theorem (for more details see here), we can compute the gradient based on two steps:

1. For each $(x, y) \in B$, solve the inner maximization problem (i.e., compute an adversarial example)
$$\delta^*(x) = \arg\max_{\delta\in\Delta} \ell(h_\theta(x+\delta)), y) \tag{3}$$

2. Compute the gradient of the empirical adversarial risk, and update
$$\theta := \theta - \frac{\alpha}{|B|} \sum_{(x,y)\in B} \nabla_\theta \ell(h_\theta(x+\delta^*(x)), y). \tag{4}$$

This procedure has become known as *adversarial training* in the deep learning literature, and (if done correctly) it is one of the most effective empirical methods we have for training adversarially robust models. However, we should note that, in most of the settings, we are never actually performing gradient descent on the true empirical adversarial risk (as defined in (1)), precisely because we typically cannot solve the inner maximization problem optimally.

In this task, we focus on a simple setting where the model is a linear model and the task is a binary classification task. In this setting, we can solve the inner maximization *exactly*. In this case, rather than using multi-class cross entropy loss, we use the binary cross entropy, or logistic loss. In this setting, we have our hypothesis function

$$h_\theta(x) = w^T x + b \tag{5}$$

for $\theta = \{w \in \mathbb{R}^n, b \in \mathbb{R}\}$, the class labels $y \in \{+1, -1\}$, and the loss function

$$\ell(h_\theta(x), y) = \log(1 + \exp(-y \cdot h_\theta(x))) \quad (= L(y \cdot h_\theta(x)))\,. \tag{6}$$

For convenience, we define the function $L(z) = \log(1+\exp(-z))$ which we will use below when discussing how to solve optimization problems involving this loss. The semantics of this setup are that for a data point x, the classifier predicts class $+1$ with probability

$$p(y = +1|x) = \frac{1}{1 + \exp(-h_\theta(x))}. \tag{7}$$

Implementation details:

You need to implement the adversarial training function `robust_trainer` in the Task 4 notebook. You are given the data loader `loader`, the model that needs to be updated `model` (linear model, e.g.,`nn.Linear(784, 1)`), the perturbation budget `epsilon`, and the optimizer `opt`.

Report:

You need to describe how you solve the inner optimization problem in the report section of the notebook. You can also inform us about any difficulties you have faced and how you solved them.

Hints:

1. Before diving into the implementation, you need to formalize the objective function that the defender needs to solve.

2. Consider the inner maximization problem, which in this case takes the form

   $$\underset{\|\delta\| \le \epsilon}{\text{maximize}}\, \ell(w^T(x+\delta), y) \equiv \underset{\|\delta\| \le \epsilon}{\text{maximize}}\, L(y \cdot (w^T(x+\delta)+b)). \tag{8}$$

   Note that, the loss L function is a scalar function. What is the monotonicity of this function?

3. Does the inner optimal solution depend on the input x? If not, you can generate a universal perturbation for all the input images.

(Optional) Task 5: Certified defense

We have seen a few defenses and attacks. Can we somehow end this arms race? In this task, we shift our attention to *certifiable robustness*. We say a classifier has certified robustness if its prediction at any point x is verifiably constant within some set around x. More specifically, we aim to build a classifier g such that

$$g(x+\delta) = k \quad, \forall \delta \in \Delta \tag{9}$$

Directly certifying the robustness of neural networks is difficult. So, instead, we look for robust classifiers that are not neural networks. The idea is to use *randomized smoothing* which can transform any arbitrary base classifier h_θ to a new "smoothed classifier" g that is certifiably robust in the l_2-norm. We focus on a specific transformation based on Gaussian noise described as follows: When queried at x, the smoothed classifier g returns whichever class the base classifier h_θ is most likely to return when x is perturbed by isotropic Gaussian noise:

$$g(x) = \arg\max_{j \in [k]} \Pr[h_\theta(x+\epsilon)_j] \text{ where } \epsilon \sim \mathbf{N}(0, \sigma^2 \mathbf{I}). \tag{10}$$

Recall that $h_\theta(x+\epsilon)_j$ means the j-th elements of the vector $h_\theta(x)$, reflecting the probability of predicting x as label j.

The result in Cohen et al. [2019] shows that smoothed classifier g is robust around x within the l_2 radius $\sigma^2(\phi^{-1}(p_A) - \phi^{-1}(p_B))$, where ϕ^{-1} is the inverse of the standard Gaussian CDF. The theorem is shown as follows:

Theorem 1 (Theorem 1 in Cohen et al. [2019]) *Let $h_\theta : \mathcal{X} \to \mathbb{R}^k$ be any deterministic or random function, and let $\epsilon \sim \mathbf{N}(0, \sigma^2 \mathbf{I})$. Let g be defined as in Equation (10). Suppose $c_A \in [k]$ and $p_A, p_B \in [0,1]$ satisfy:*

$$\Pr[h_\theta(x+\epsilon)_{c_A}] \geq p_A \geq p_B \geq \max_{c \neq c_A} \Pr[h_\theta(x+\epsilon)_c] \tag{11}$$

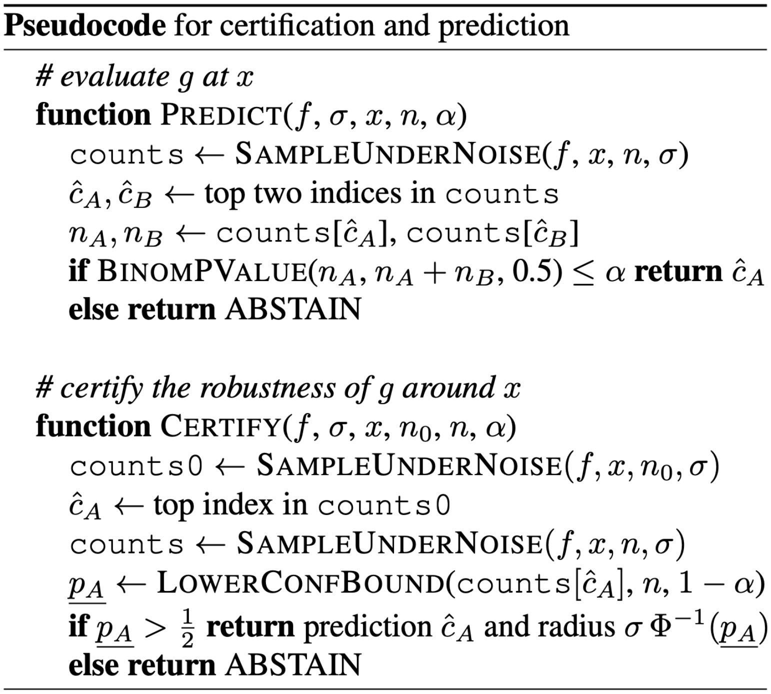
Then $g(x+\delta) = c_A$ for all $\|\delta\|_2 \leq R$, where

$$R = \sigma^2(\phi^{-1}(p_A) - \phi^{-1}(p_B)). \tag{12}$$

To certify the robustness of g, we need to know p_A and p_B, which can not be obtained in practice. To solve this problem, we can estimate p_A and p_B using Monte Carlo methods. The pseudocode of the algorithm is shown in Figure 6, where f is the base classifier h_θ and n (and n_0) is the number of Monte Carlo samples for estimation (and selection).

Implementation details

In this task, you need to implement the randomized smoothing for a given trained model. You need to implement function `certify` and `predict` in the `Guassian_smoothed_classifier` class in the Task 5 notebook.



Pseudocode for certification and prediction

```
  # evaluate $g$ at $x$
  function PREDICT($f, \sigma, x, n, \alpha$)
      counts ← SAMPLEUNDERNOISE($f, x, n, \sigma$)
      $\hat{c}_A, \hat{c}_B$ ← top two indices in counts
      $n_A, n_B$ ← counts[$\hat{c}_A$], counts[$\hat{c}_B$]
      if BINOMPVALUE($n_A, n_A + n_B$, 0.5) $\leq \alpha$ return $\hat{c}_A$
      else return ABSTAIN

  # certify the robustness of $g$ around $x$
  function CERTIFY($f, \sigma, x, n_0, n, \alpha$)
      counts0 ← SAMPLEUNDERNOISE($f, x, n_0, \sigma$)
      $\hat{c}_A$ ← top index in counts0
      counts ← SAMPLEUNDERNOISE($f, x, n, \sigma$)
      $\underline{p_A}$ ← LOWERCONFBOUND(counts[$\hat{c}_A$], $n$, $1 - \alpha$)
      if $\underline{p_A} > \frac{1}{2}$ return prediction $\hat{c}_A$ and radius $\sigma \, \Phi^{-1}(\underline{p_A})$
      else return ABSTAIN
```

Figure 6: Pseudocode for certifying robustness (Cohen et al. [2019])

Test Your Code

To ensure that your attack algorithms can be executed correctly without errors, each notebook comes with a testing block. Please do not change anything in the testing section besides changing the hyper-parameters in the last testing cell, unless you are told to do so (e.g. copy and paste your previous algorithm).

References

Jeremy Cohen, Elan Rosenfeld, and Zico Kolter. Certified adversarial robustness via randomized smoothing. In *International Conference on Machine Learning*, pages 1310–1320. PMLR, 2019.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*, 2017.

Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *2016 IEEE European symposium on security and privacy (EuroS&P)*, pages 372–387. IEEE, 2016.

Cihang Xie, Jianyu Wang, Zhishuai Zhang, Zhou Ren, and Alan Yuille. Mitigating adversarial effects through randomization. *arXiv preprint arXiv:1711.01991*, 2017.