CIS 6261: Trustworthy Machine Learning (Spring 2023)
Homework 1 — Adversarial Examples

Name: Anol Kurian Vadakkeparampil

March 1, 2023

This is an individual assignment. Academic integrity violations (i.e., cheating, plagiarism) will be reported to SCCR! The official CISE policy recommended for such offenses is a course grade of E. Additional sanctions may be imposed by SCCR such as marks on your permanent educational transcripts, dismissal or expulsion.

Reminder of the Honor Pledge: On all work submitted for credit by Students at the University of Florida, the following pledge is either required or implied: *"On my honor, I have neither given nor received unauthorized aid in doing this assignment."*

Instructions

Please read the instructions and questions carefully. Write your answers directly in the space provided. Compile the tex document and hand in the resulting PDF.

In this assignment you will explore adversarial examples in Python. Use the code skeleton provided and submit the completed source file(s) alongside with the PDF.[1] *Note: bonus points you get on this homework *do* carry across assignments/homework.*

Assignment Files

The assignment archive contains the following Python source files:

- `hw.py`. This file is the main assignment source file.
- `nets.py`. This file defines the neural network architectures and some useful related functions.
- `attacks.py`. This file contains attack code used in the assignment.

Note: You are encouraged to carefully study the provided files. This may help you successfully complete the assignment.

[1] You should use Python3 and Tensorflow 2. You may use HiPerGator or your own system. This assignment can be completed with or without GPUs.

Problem 0: Training a Neural Net for MNIST Classification (10 pts)

In this problem, you will train a neural network to do MNIST classification. The code for this problem uses the following command format.

```
python3 hw.py problem0 <nn_desc> <num_epoch>
```

Here `<nn_desc>` is a neural network description string (no whitespaces). It can take two forms: `simple,<num_hidden>,<l2_reg_const>` or `deep`. The latter specifies the deep neural network architecture (see `get_deeper_classifier()` in `nets.py` for details), whereas the former specifies a simple neural network architecture (see `get_simple_classifier()` in `nets.py` for details) with one hidden layer with `<num_hidden>` neurons and an L_2 regularization constant of `<l2_reg_const>`. Also, `<num_epoch>` is the number of training epochs.

For example, suppose you run the following command.

```
python3 hw.py problem0 simple,64,0.001 100
```

This will train the target model on MNIST images for 100 epochs.[2] The target model architecture is a neural network with a single hidden layer of 64 neurons which uses L_2 regularization with a constant of 0.001.[3] (The loss function is the categorical cross-entropy loss.)

1. (5 pts) Run the following command:

   ```
   python3 hw.py problem0 simple,128,0.01 20
   ```

   This will train the model and save it on the filesystem. Note that '`problem0`' is used to denote training. The command line for subsequent problems (`problem1`, `problem2`, etc.) will load the model trained.

   Before you can run the code you need to put in your UFID at the beginning of the `main()` function in `hw.py`.

2. (5 pts) What is the training accuracy? What is the test accuracy? Is the model overfitted?

   Train accuracy: 96.01%. Test accuracy: 95.45%. The model does not seem to be significantly overfitted, the difference between the training accuracy and the test accuracy is not very large, and both accuracies are high

[2] Each MNIST image is represented as an array of $28 \cdot 28 = 784$ pixels, each taking a value in $\{0, 1, \ldots, 255\}$.
[3] By default, the code will provide detailed output about the training process and the accuracy of the target model.

Problem 1: Mysterious Attack (50 pts)

For this problem, you will study an unknown attack that produces adversarial examples. This attack is called the gradient noise attack. You will look at its code and run it to try to understand how it works. (The code is in `gradient_noise_attack()` which is located in `attacks.py`.)

This attack is already implemented, so you will only need to run it and answer questions about the output.

However, before you can run the attack you will need to implement `gradient_of_loss_wrt_input()` found in `nets.py`. This function computes the gradient of the loss function with respect to the input. We will use it for the subsequent problems, so make sure you implement it correctly!

To run the code for this problem, use the following command.

```
python3 hw.py problem1 <nn_desc> <input_idx> <alpha>
```

Here `<input_idx>` is the input (benign) image that the attack will create an adversarial example from and `<alpha>` is a non-negative integer parameter used by the attack. The code will automatically load the model from file, so you need to have completed problem0 first!

1. (5 pts) Before we can reason about adversarial examples, we need a way to quantify the distortion of an adversarial perturbation with respect to the original image. Propose a metric to quantify this distortion as a single real number.[4] Explain your choice.

   A widely adopted approach for measuring the distortion of an adversarial perturbation is to use the L_p norm between the original image and the perturbed image. The value of p can influence the metric's sensitivity to minor changes in the image.

   To compute the Lp norm between the original image xin and the adversarial perturbation x_{adv}, this implementation employs `tf.norm()` from TensorFlow. The default value of p is 2, which corresponds to the Euclidean norm..

   Locate the incomplete definition of the `distortion`() function in `hw.py`, and implement your proposed metric. What is the range of your distortion metric?

   The range of the distortion metric depends on the value of p. When $p = 2$, the distortion metric is non-negative and its range is from 0 (no distortion) to infinity (in the limit where the perturbation becomes unbounded). However, for other values of p, the range of the distortion metric can vary. For instance, when $p = 1$, the range of the distortion metric ranges from 0 to the maximum possible value of the sum of absolute differences between the pixel values of the original and perturbed images.

2. (10 pts) Before we can run the attack, you need to implement `gradient_of_loss_wrt_input()` located in `nets.py`. For this, you can use Tensorflow's `GradientTape`. Follow the instructions in the comments and fill in the implementation (about 5 lines of code). Make sure this is implemented correctly and copy-paste your code below.

   To compute the gradient of the loss with respect to the input tensor 'x' in a TensorFlow 2.x model, you can use the following function:

```
def gradient_of_loss_wrt_input(model, x, y):
    x = tf.convert_to_tensor(x, dtype=tf.float32)
    # convert to tensor
    y = tf.convert_to_tensor(y.reshape((1, -1)), dtype=tf.float32)
    # convert to tensor
    with tf.GradientTape() as tape:
        tape.watch(x)
        # Watch the input tensor 'x' to compute its gradient
        y_pred = model(x)
        # Forward pass to obtain the predicted class probabilities
```

[4] The specific metric you implement is your choice and there many possible options, but you probably want to ensure that two identical images have a distortion of 0 and that any two different images have a distortion larger than 0, with the larger the difference between the images the larger the distortion value.

```
        loss = tf.keras.losses.CategoricalCrossentropy()(y, y_pred)
        # Compute the loss

    # Compute the gradient of the loss with respect to 'x'
    grad_x = tape.gradient(loss, x)

    return grad_x
```

3. (15 pts) Now, let's run the attack using the following command with various input images and alphas.

```
python3 hw.py problem1 simple,128,0.01 <input_idx> <alpha>
```

Note: it is important than the architecture match what you ran for Problem 0. (The code uses these arguments to locate the model to load.)

For example, try:

```
python3 hw.py problem1 simple,128,0.01 0 2
python3 hw.py problem1 simple,128,0.01 0 15
python3 hw.py problem1 simple,128,0.01 1 4
python3 hw.py problem1 simple,128,0.01 1 40
python3 hw.py problem1 simple,128,0.01 2 8
python3 hw.py problem1 simple,128,0.01 3 1
python3 hw.py problem1 simple,128,0.01 4 1
python3 hw.py problem1 simple,128,0.01 5 4
python3 hw.py problem1 simple,128,0.01 6 9
```

If you have implemented the previous function correctly, the code will plot the adversarial examples (see `gradient_noise.png`) and print the distortion according to your proposed metric.

Produce adversarial examples for at least four different input examples and two values for the alpha parameter. Paste the plots here. (Use minipage and subfigure to save space.)

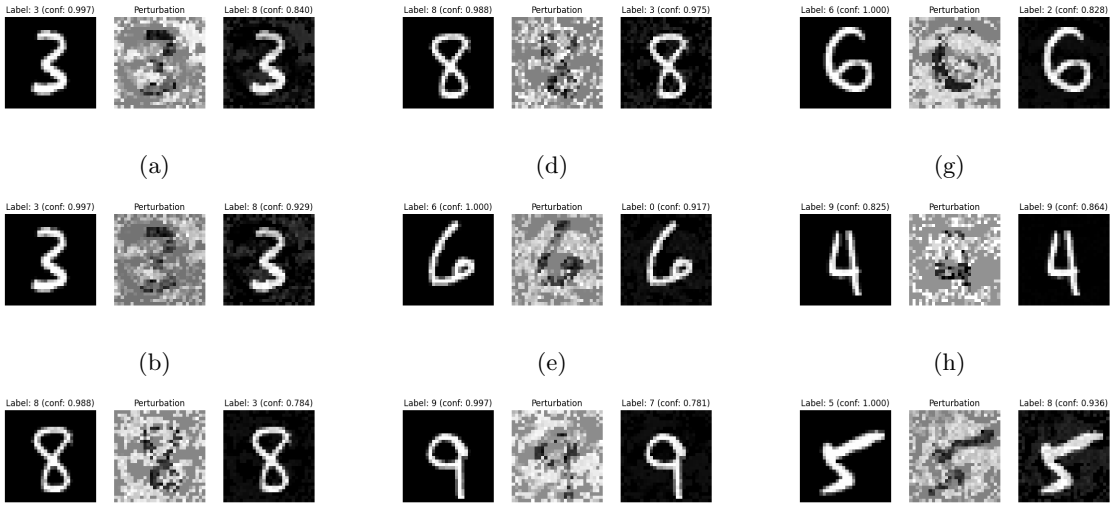

Figure 1

Do you observe any failures? Do successful adversarial examples look like the original image? What do think is the purpose of the parameter alpha?

The model fails to recognize certain images, as shown in Figure 1.h. However, no attack failures were observed during the attacks that were run. The attacks were successful in causing the model to misclassify all images. But the misclassified images in the original model did not seem to have changed their label after the attack.

To the human eye, the adversarial examples do look significantly similar to the original image.

*The alpha parameter in the gradient_noise_attack function controls the magnitude of the perturbation added to the input image at each iteration of the attack. Specifically, the perturbation is defined as alpha * r * tf.sign(grad_vec), where r is a random vector and grad_vec is the gradient of the loss with respect to the input image. The sign of grad_vec determines the direction of the perturbation, while alpha scales its magnitude.*

The purpose of alpha is to control the strength of the attack. A larger value of alpha results in a stronger perturbation, which can potentially cause the model to misclassify the input image. However, a very large value of alpha can cause the perturbation to be too strong and result in a distorted image that does not resemble the original. On the other hand, a small value of alpha can make the perturbation too weak and not effective in fooling the model.

Therefore, choosing alpha requires balancing the need to produce a strong perturbation with the desire to maintain a certain level of visual similarity between the original image and the adversarial example. The optimal value of alpha may depend on the specific attack scenario and the characteristics of the model being attacked.

4. (15 pts) Now, let's look into the code of the attack (`gradient_noise_attack()` located in `attacks.py`) and try to understand how it works.

   First focus on lines 39 to 44 where the perturbation is created and added to the adversarial example. How is the perturbation made and how does it use the gradient of the loss with respect to the input?

   Lines 39 to 44 generate a stochastic disturbance using a uniform distribution ranging from 0 to 1. The magnitude of the disturbance is determined by the parameter alpha, which governs the extent of the perturbation. The direction of the disturbance is determined by the sign of the gradient of the loss with respect to the input, indicating the direction in which the loss increases. Consequently, the perturbation can shift the input in a direction that is more prone to inducing misclassification.

   The code uses `tf.clip_by_value()`. What is the purpose of this function and why is it used by the attack?

   The tf.clip_by_value() function on line 47 is employed to restrict the pixel values of the adversarial example within the acceptable range of 0 to 255 in LaTeX. This is an essential step to ensure that the adversarial example retains its validity as an image.

   Now let's look at lines 50 to 57. Is the attack targeted or untargeted? What is the purpose of `target_class_number`? How does the attack terminate and why?

   The attack is focused on finding an adversarial example that is misclassified as a specific target class. This target class is determined by the label that is most likely to be incorrect based on the model's predictions. To specify this target class, a target class number is used.

   The attack will continue until the terminate_fn() function returns True. This indicates that the attack has successfully found an adversarial example that is misclassified as the target class. The function takes four inputs: the original input, the current adversarial example, the target class number, and the number of iterations that have been performed so far. The function can be used to set a limit on the maximum number of iterations or to stop the attack if the distortion between the original input and the adversarial example exceeds a certain threshold.

5. (5 pts) Finally let's look at the lines 35 to 37 (the if branch). What functionality is implemented by this short snippet of code? Give a reason why doing this is a good idea.

   The if branch checks if the sum of the absolute values of the elements in the gradient vector is less than a small value $1sf1$. If this condition is satisfied, it means that the gradient vector is close to zero or too small to be useful in generating a perturbation that will move the input

away from the current prediction. In this case, the code replaces the gradient vector with a small random perturbation, with each element drawn from a normal distribution with a mean of zero and a standard deviation of 0.0001*.*

Doing this is a good idea because when the gradient is small or close to zero, adding a random perturbation can help explore the space of possible inputs and prevent the attack from getting stuck in a local minimum or maximum. This can make the attack more effective in finding adversarial examples.

Problem 2: Strange Predictions (10 pts)

In this problem, we will look at strange behavior of neural nets using our MNIST classification model. Specifically, we will study the behavior of the model when given random images as input.

1. (5 pts) Locate the `random_images`() function in main of `hw.py`. The purpose of this function is to generate random images in the input domain of MNIST. Each image is represented as a 1×784 array of pixels (integers) with each pixel taking a value in $\{0, 1, \ldots, 255\}$.

   Fill in the code to draw random images with independent pixel values selected uniformly in $\{0, 1, \ldots, 255\}$. Make sure you return image data with shape that match the `size` parameter. Once you have implemented this, run the following command.

   ```
   python3 hw.py problem2 simple,128,0.01
   ```

   The code will plot the distribution of predictions for random images (estimated over a large number of samples). Paste the plot here. What does the distribution look like? Is this expected or unexpected?

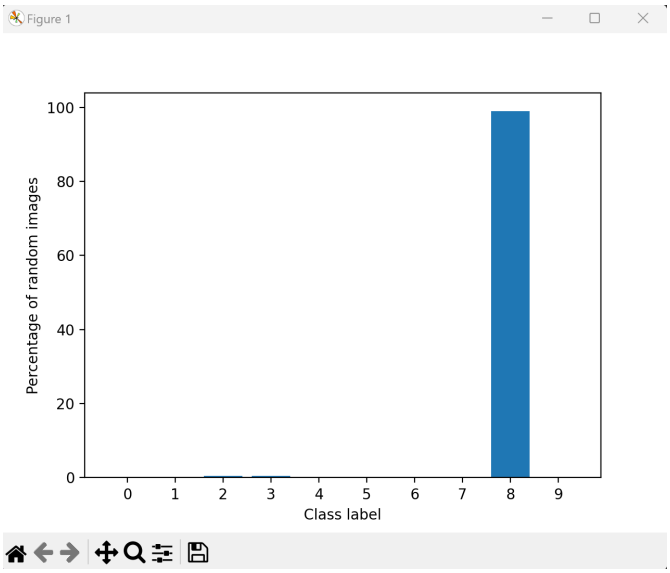

Figure 2

The model's predictions on a set of 1000 images generated randomly from the MNIST input domain show a strong skew towards the digit 8, with very few images classified as 2 or 3.

This is surprising because it implies that the model has a bias towards recognizing only a few specific digits, and struggles to recognize others. Despite this, the model's overall accuracy on the test set is high (95.45%), indicating that it can recognize digits from 0 to 9 with reasonable accuracy. However, the skewed distribution of predictions on the randomly generated images suggests that the model may not be resilient to changes in the input images and may be overfitted to particular patterns in the training data that do not represent the whole input domain.

2. (5 pts) Is there a relationship between the previous observation and the failure(s) you observed in problem1?

   In my opinion, the connection between these two is unclear. It appears that the issue with the model is linked to a bias towards specific numbers, such as 8, while the problem in the first scenario appears to be random.

Problem 3: Iterative FGSM Attacks (15 pts)

In this problem, we will study iterative FGSM attacks.

1. (10 pts) Locate the `do_untargeted_fgsm()` function in `attacks.py`. Implement the body of the function according to the instructions in the comments. You can also refer to the course slides. Note that the version you have to implement is untargeted.

   Make sure it is implemented correctly and copy-paste your code for this function below.

```
def do_untargeted_fgsm(model, in_x, y, alpha):

    grad_vec = nets.gradient_of_loss_wrt_input(model, in_x, y)
    # Calculate the perturbation using the sign of the gradient to
    # increase the loss on 'in_x' for the true label
    # calculate perturbation
    perturbation = alpha * tf.sign(grad_vec)

    # add perturbation to input image
    adv_x = in_x + perturbation

    # clip the adversarial example to ensure it is a valid image
    adv_x = tf.clip_by_value(adv_x, clip_value_min=0, clip_value_max=255)

    return adv_x
```

   You can familiarize yourself with the code of `run_iterative_fgsm()` which calls `iterative_fgsm()` function in `attacks.py`. Both are already implemented for you. (You can also use the provided `plot_adversarial_example()` to help debug your code.)

2. (5 pts) Now, let's run the attack using the following command with a specific number of adversarial examples (e.g., 100 or 200) and perturbation magnitude ϵ (e.g., $\epsilon = 20$). (Depending on the parameters you choose, it could take a few minutes.)

```
python3 hw.py problem3 simple,128,0.01 <num_adv_samples> <eps>
```

   The code will save the adversarial examples created to a file. It will also evaluate the success rate of the attack using the `evaluate_attack()` function located in `attacks.py`.

   What is the success rate of the untargeted attack? Explain what the benign accuracy and adversarial accuracy measure.

   Benign acc: 99.0%, Adversary acc: 46.0%

   *Success rate of untargeted attack = (number of misclassified adversarial examples / number of adversarial examples generated) * 100% = (54 / 100) * 100% = 54.0%*

   Benign Accuracy refers to the accuracy of a model when it is tested on a set of examples that are representative of the real-world data the model is designed to handle.

   On the other hand, Adversarial Accuracy refers to the accuracy of a model when it is tested on examples that are specifically designed to fool the model into making incorrect predictions.

Problem 4: Randomized Smoothing (15 pts)

In this problem, we will implement a defense based on randomized smoothing. The code for this problem should be invoked as follows:

```
python3 hw.py problem4 simple,128,0.01 <eps> <sigma_str>
```

where `<eps>` denotes ϵ the magnitude of the perturbation (this is necessary to load files of adversarial examples saved in problem3) and `<sigma_str>` is a comma-delimited list of values of σ for randomized smoothing. For example: "`1,5,10,20`" means to perform randomized smoothing for $\sigma = 1$, then again for $\sigma = 5$, then $\sigma = 10$, and finally $\sigma = 20$.

1. (10 pts) Locate `randomized_smoothing_predict_fn`() in `hw.py`. You will need to implement Gaussian noise addition. (The rest of code that is provided already does averaging of predictions.) Follow the instruction in comments and refer to the course slides.

   Run the code for $\epsilon = 20$ and with a reasonable list of sigma values. Paste your output below and explain how you interpret the results printed out when you run the code.

   How effective is the defense? How many adversarial examples did you evaluate the defense on?

   ```
   PS C:\Users\anolk\Desktop\TML_HW> python3 hw.py problem4
   simple,128,0.01 20 1,5,10,20,30,40,50
   2023-03-01 14:07:32.650045: I tensorflow/core/platform/cpu_feature_guard.cc:193]
   This TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN)
   to use the following CPU instructions in performance-critical operations:  AVX AVX2
   To enable them in other operations, rebuild TensorFlow with the appropriate compiler
   flags.
   ----- UFID: 56268544 ; r: 0.840579
   Loaded mnist data; shape: (60000, 28, 28) [y: (60000,)],
   test shape: (10000, 28, 28) [y: (10000,)]
   Loaded model from file (C:\Users\anolk\Desktop\TML_HW\models/simple,128,0.01)
   -- [4EB5848E2124765D].
   [RS] Untargeted FGSM attack eval --- benign acc: 96.0%, adv acc: 43.0%
   [eps=20.0, sigma: 1, noise: gaussian]
   [RS] Untargeted FGSM attack eval --- benign acc: 96.0%, adv acc: 46.0%
   [eps=20.0, sigma: 5, noise: gaussian]
   [RS] Untargeted FGSM attack eval --- benign acc: 96.0%, adv acc: 49.5%
   [eps=20.0, sigma: 10, noise: gaussian]
   [RS] Untargeted FGSM attack eval --- benign acc: 96.5%, adv acc: 54.5%
   [eps=20.0, sigma: 20, noise: gaussian]
   [RS] Untargeted FGSM attack eval --- benign acc: 96.0%, adv acc: 55.0%
   [eps=20.0, sigma: 30, noise: gaussian]
   [RS] Untargeted FGSM attack eval --- benign acc: 92.5%, adv acc: 50.5%
   [eps=20.0, sigma: 40, noise: gaussian]
   [RS] Untargeted FGSM attack eval --- benign acc: 89.5%, adv acc: 47.5%
   [eps=20.0, sigma: 50, noise: gaussian]
   [0.96, 0.96, 0.96, 0.965, 0.96, 0.925, 0.895]
   [0.43, 0.46, 0.495, 0.545, 0.55, 0.505, 0.475]
   ```

   The defense does not seem to be very effective since accuracy was not increased tremendously as seen in the output above. It increased from a low of 43% to a max of 55% for sigma value of 30.

   The defense was evaluated for 200 adversarial examples at epsilon 20.

   Note: you may need to re-run problem3 to generate sufficiently many adversarial examples so you can make sound conclusions when you answer this question.

2. (5 pts) Let's explore the relationship induced by σ between the two kinds of accuracies. For this you should run the code again to obtain data for sufficiently many different sigma values (e.g., 0 to 100).

Plot a figure or create a table to show this relationship. You can add your code at the end of the problem4 `if` branch in `hw.py`. The two accuracies are saved in `benign_accs` and `adv_accs`.

Paste the plot/table below and briefly comment on the relationship.

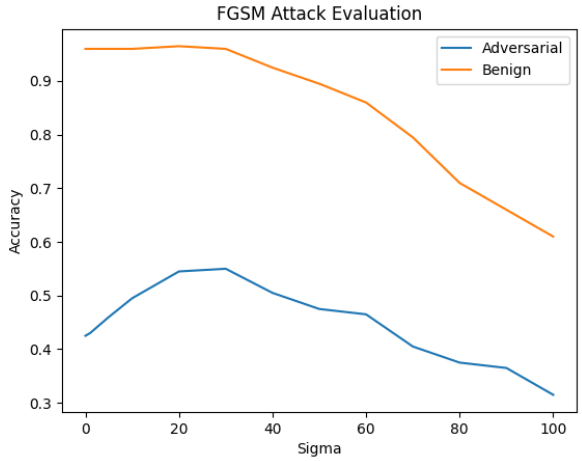

Figure 3

The smoothening defense was successful to a certain point after which the accuracy for both adversarial and benign samples started falling.

3. ***[Bonus]*** (5 pts) Implement randomized smoothing with Laplace noise. You should add your code inside `randomized_smoothing_predict_fn`() and switch between the two noises using `noise_type` (which is passed as an optional command line argument). For passing the Laplace lambda parameter reuse sigma. Make sure that the Gaussian noise version still works as intended.

   Paste the plot/table of randomized smoothing with Laplace noise below. Which type of noise is more effective against the iterative FGSM attack? (Justify your reasoning.)

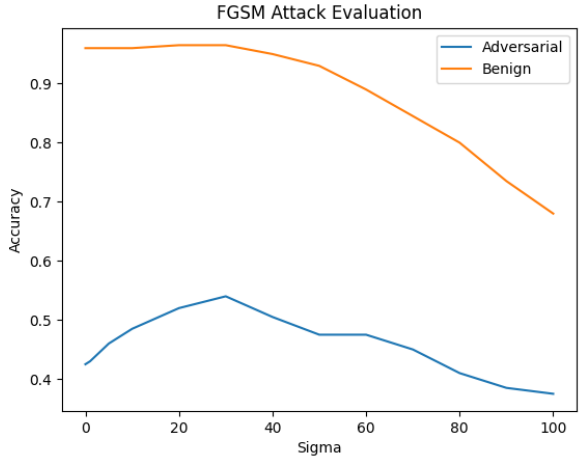

Figure 4

For our samples, Gaussian noise seems to do a better job at handling adversarial attacks.

A smoothing method known as Gaussian noise is extensively utilized in computer vision and image processing applications. Its effectiveness lies in its ability to blur the input data and minimize the influence of high-frequency features that attackers may exploit. By incorporating

random values sampled from a Gaussian distribution into the input data, Gaussian noise produces a smooth and continuous noise that retains the statistical characteristics of the underlying data.

[Bonus] Problem 5: Transferability (5 pts)

For this (bonus) problem you will train a new model (different from the one your trained in problem 0 but trained on the same data) and evaluate the transferability of adversarial examples produced by iterative FGSM (problem3) on this new model.

1. (5 pts) Run the attack on the new model. Hint: there is a way to do this without changing/adding a single line of code (can you think of it?). If you cannot, feel free to use the `problem5 if-branch` in `hw.py` to put your additional code.

   First explain briefly your methodology for this question. What is the architecture of the other model that you chose? How did the attack perform on the original model? (Include details below.)

   > *I chose the 'deep' neural network model due to its deeper architecture, which may allow it to learn more complex representations of the data. Additionally the code for it is already given and I do not need to make any changes.*
   >
   > *To choose an alternative architecture, my methodology would involve considering a range of factors, such as the complexity of the architecture, the number of layers, the type of layers used, and the activation functions used. I would also consider the performance of the architecture on the task at hand, as well as its robustness to adversarial attacks.*
   >
   > *The attack on the original model had a relatively high success rate on adversarial examples, with an adversarial accuracy of 42.5%. This means that the attack was successful on 42.5% of the adversarial examples generated, while the model correctly classified 96.0% of the benign examples.*

   Now include details about success rate of the attack on the new model. What do you conclude about transferability? (Justify your answer.)

   > *The second alternative model has a deeper architecture compared to the original model, which may make it more resistant to adversarial attacks. However, the success rate of the attack on this model is still lower than its benign accuracy, indicating that it is not completely robust to adversarial examples.*
   >
   > *The adversarial accuracy of the new model is 77.5%, while its benign accuracy is 98.0%. This means that the model was successful on 77.5% of the adversarial examples generated, while the model correctly classified 98.0% of the benign examples.*
   >
   > *Regarding transferability, the low adversarial accuracy on the new model suggests that the attack is not transferable to the new model. This means that the adversarial examples generated for the original model are not effective on the new model, and that the new model has learned different decision boundaries that are more robust to adversarial attacks. This is a positive result, as it suggests that the new model is more likely to generalize to new, unseen data.*