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

Name: Yashaswini Kondakindi

March 2, 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.

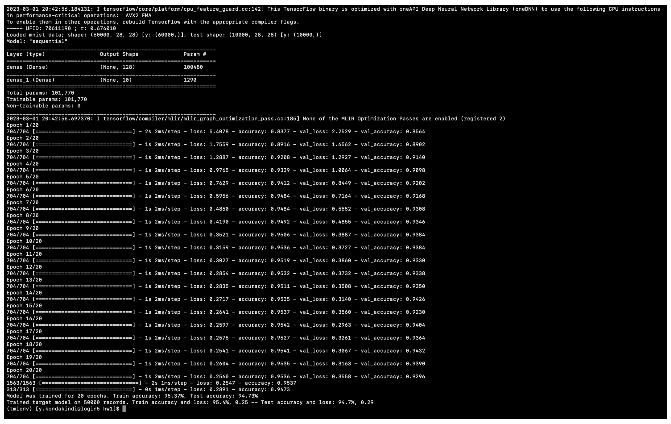
```
python3 hw.py problem0 simple,64,0.001 100
```

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.



```
2023-03-01 20:42:56.184131: I tensorflow/core/platform/cpu_feature_guard.cc:142] This TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions
in performance-critical operations:  AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
----- UFID: 70611190 ; r: 0.676010
Loaded mnist data; shape: (60000, 28, 28) [y: (60000,)], test shape: (10000, 28, 28) [y: (10000,)]
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 128)               100480
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1290
=================================================================
Total params: 101,770
Trainable params: 101,770
Non-trainable params: 0
_________________________________________________________________
2023-03-01 20:42:56.697370: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:185] None of the MLIR Optimization Passes are enabled (registered 2)
Epoch 1/20
704/704 [==============================] - 2s 2ms/step - loss: 5.4078 - accuracy: 0.8377 - val_loss: 2.2529 - val_accuracy: 0.8564
Epoch 2/20
704/704 [==============================] - 1s 2ms/step - loss: 1.7559 - accuracy: 0.8916 - val_loss: 1.6562 - val_accuracy: 0.8902
Epoch 3/20
704/704 [==============================] - 1s 2ms/step - loss: 1.2887 - accuracy: 0.9208 - val_loss: 1.2927 - val_accuracy: 0.9140
Epoch 4/20
704/704 [==============================] - 1s 2ms/step - loss: 0.9765 - accuracy: 0.9339 - val_loss: 1.0064 - val_accuracy: 0.9098
Epoch 5/20
704/704 [==============================] - 1s 2ms/step - loss: 0.7629 - accuracy: 0.9412 - val_loss: 0.8449 - val_accuracy: 0.9202
Epoch 6/20
704/704 [==============================] - 1s 2ms/step - loss: 0.5956 - accuracy: 0.9484 - val_loss: 0.7164 - val_accuracy: 0.9168
Epoch 7/20
704/704 [==============================] - 1s 2ms/step - loss: 0.4850 - accuracy: 0.9484 - val_loss: 0.5552 - val_accuracy: 0.9308
Epoch 8/20
704/704 [==============================] - 1s 2ms/step - loss: 0.4190 - accuracy: 0.9492 - val_loss: 0.4855 - val_accuracy: 0.9346
Epoch 9/20
704/704 [==============================] - 1s 2ms/step - loss: 0.3521 - accuracy: 0.9506 - val_loss: 0.3887 - val_accuracy: 0.9384
Epoch 10/20
704/704 [==============================] - 1s 2ms/step - loss: 0.3159 - accuracy: 0.9536 - val_loss: 0.3727 - val_accuracy: 0.9384
Epoch 11/20
704/704 [==============================] - 1s 2ms/step - loss: 0.3027 - accuracy: 0.9519 - val_loss: 0.3860 - val_accuracy: 0.9330
Epoch 12/20
704/704 [==============================] - 1s 2ms/step - loss: 0.2854 - accuracy: 0.9532 - val_loss: 0.3732 - val_accuracy: 0.9338
Epoch 13/20
704/704 [==============================] - 1s 2ms/step - loss: 0.2835 - accuracy: 0.9511 - val_loss: 0.3508 - val_accuracy: 0.9350
Epoch 14/20
704/704 [==============================] - 1s 2ms/step - loss: 0.2717 - accuracy: 0.9535 - val_loss: 0.3140 - val_accuracy: 0.9426
Epoch 15/20
704/704 [==============================] - 1s 2ms/step - loss: 0.2641 - accuracy: 0.9537 - val_loss: 0.3560 - val_accuracy: 0.9230
Epoch 16/20
704/704 [==============================] - 1s 2ms/step - loss: 0.2597 - accuracy: 0.9542 - val_loss: 0.2963 - val_accuracy: 0.9404
Epoch 17/20
704/704 [==============================] - 1s 2ms/step - loss: 0.2575 - accuracy: 0.9527 - val_loss: 0.3261 - val_accuracy: 0.9364
Epoch 18/20
704/704 [==============================] - 1s 2ms/step - loss: 0.2541 - accuracy: 0.9541 - val_loss: 0.3067 - val_accuracy: 0.9432
Epoch 19/20
704/704 [==============================] - 1s 2ms/step - loss: 0.2604 - accuracy: 0.9535 - val_loss: 0.3163 - val_accuracy: 0.9390
Epoch 20/20
704/704 [==============================] - 1s 2ms/step - loss: 0.2560 - accuracy: 0.9536 - val_loss: 0.3558 - val_accuracy: 0.9296
1563/1563 [==============================] - 2s 1ms/step - loss: 0.2547 - accuracy: 0.9537
313/313 [==============================] - 0s 1ms/step - loss: 0.2891 - accuracy: 0.9473
Model was trained for 20 epochs. Train accuracy: 95.37%, Test accuracy: 94.73%
Trained target model on 50000 records. Train accuracy and loss: 95.4%, 0.25 -- Test accuracy and loss: 94.7%, 0.29
(tmlenv) [y.kondakindi@login5 hw1]$
```

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

   Training accuracy: 95.37% . Testing accuracy: 94.73% .

   Based on the accuracy values, it seems like the model is performing well on both training and testing datasets, but there is a slight gap between the training and testing accuracy. The training accuracy is slightly higher than the testing accuracy, which suggests that the model is slightly overfitting the training data.

   However, the difference in accuracy between the training and testing data is not significant, so it is possible that the model is not overfitting too much.

[2]Each MNIST image is represented as an array of $28 \cdot 28 = 784$ pixels, each taking a value in $\{0, 1, \dots, 255\}$.
[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.

   One possible metric to quantify the distortion of an adversarial perturbation with respect to the original image is the Mean Squared Error (MSE) between the original image and the adversarial image. The MSE is a measure of the average squared difference between the pixel values of the two images. The MSE has several desirable properties as a distortion metric, including that it is always non-negative, with a value of 0 indicating that the two images are identical, and a larger value indicating a greater difference between the images. Additionally, the MSE is sensitive to both small and large changes in the pixel values, so it can capture a wide range of adversarial perturbations.

   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 MSE measures the average of the squared differences between the original and the distorted signal. The range of MSE is from 0 to infinity, where a lower value indicates less distortion. The range of MSE would depend on the number of pixels in the image and the color depth of the pixels. The range of the distortion metric depends on the range of pixel values in the image. Eight-bit images contain 256 possible values for each pixel, as every pixel is represented by 8 bits. For an 8-bit image, the pixel values range from 0 to 255. Therefore, the distortion metric ranges from 0 to 255.

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.

```
def gradient_of_loss_wrt_input(model, x, y):
    x = tf.convert_to_tensor(x, dtype=tf.float32)
    y = tf.convert_to_tensor(y.reshape((1, -1)), dtype=tf.float32)
    with tf.GradientTape() as tape:
      tape.watch(x)
      logits = model(x)
      loss = tf.keras.losses.categorical_crossentropy(y, logits)
    grad = tape.gradient(loss, x)
    return grad.numpy()
```

[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.

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.)

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 alpha parameter controls the strength of the perturbation applied to the input image during the generation of adversarial examples. As alpha increases, the amount of perturbation applied to the image increases, making it harder for the model to classify the image correctly. Alpha values greater than 50 result in an exception error, preventing the attack from being carried out. On the basis of the above generated images for different alpha values, it is clear that the images were generated correctly, but their clarity varies based on the alpha value. Even after perturbation, most images appear exactly the same. The parameter alpha indicates how much perturbation occurred. The more alpha there is, the greater the amount of perturbation, making the output image distorted and unclear to classify, resulting in decreased accuracy.

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.

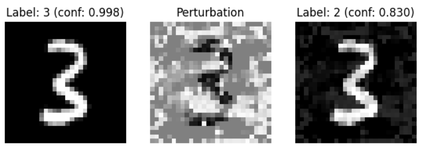
First focus on lines 39 to 44 where the perturbation is created and added to the adversarial example. How is the perturbation made and how does it use the gradient of the loss with respect to the input?

The alpha parameter controls the magnitude of the perturbation added to the input image. A higher value of alpha results in a larger perturbation and a more significant change in the input image. The resulting tensor perturb is then added to the original input image. This operation perturbs the input image in the direction that maximizes the loss function, resulting in an adversarial example that is mis-classified by the neural network.

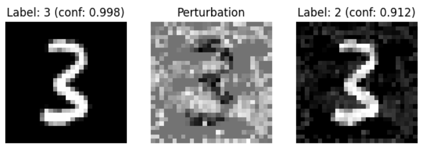
The code uses `tf.clip_by_value()`. What is the purpose of this function and why is it used by the attack?

The `tf.clip_by_value()` function is commonly used in deep learning models to ensure that the values of a tensor are within a specific range. In the context of generating adversarial examples, it's important to ensure that the pixel values of the perturbed image are within the valid range of pixel values for the input image, which is typically 0 to 255 for 8-bit images.

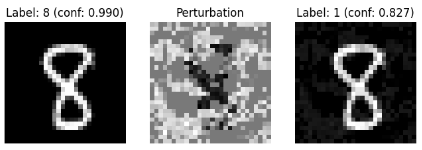
Now let's look at lines 50 to 57. Is the attack targeted or untargeted? What is the purpose of `target_class_number`? How does the attack terminate and why?



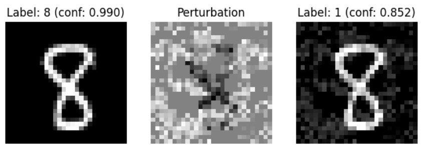
(a) $imageIndex = 0; alpha = 2$



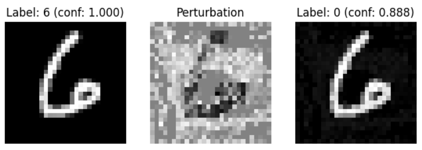
(b) $imageIndex = 1; alpha = 15$



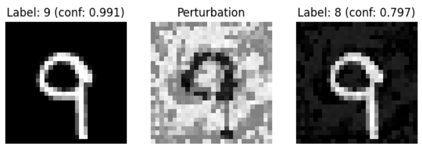
(c) $imageIndex = 1; alpha = 4$



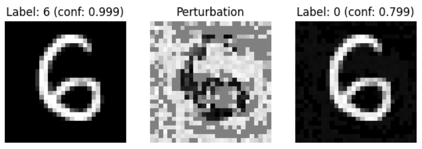
(d) $imageIndex = 1; alpha = 40$



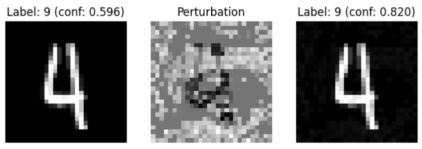
(e) $imageIndex = 2; alpha = 8$



(f) $imageIndex = 3; alpha = 1$



(g) $imageIndex = 4; alpha = 1$



(h) $imageIndex = 5; alpha = 4$

Figure 1: Plots for examples of adversarial attacks

The attack targets the predicted class corresponding to yflat by setting the predicted class to zero and setting targetclass to the class with the highest predicted probability in y pred. The target class number is set to the most likely incorrect class. When a certain condition is met, the attack terminates early, suggesting that the attacker has a specific goal and is trying to achieve it in the most efficient way. Using this approach, a targeted attack occurs when the iteration reaches a threshold or the model misclassifies an image as target.

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 purpose of adding random noise to the gradient vector when its sum of absolute values is less than the threshold is to prevent the optimization process from getting stuck at a local minimum or maximum. This is because when the gradient vector is close to zero, the attack may converge to a suboptimal solution, rather than finding the global optimum. By adding random noise to the gradient, the optimization process is encouraged to explore different directions and potentially find better solutions. This technique is called random restarts and is commonly used in optimization problems to avoid getting trapped in local optima.

   This code replaces the gradient vector gradvec with a small random vector obtained using tf.random, scaled by a small factor 0.0001 when the sum of absolute values is less than the threshold. The gradient vector gradvec is close to zero and adding it may not result in significant changes. As a result, the perturbation is added some randomness and variability even when the gradient vector is close to zero. Consequently, the attack won't get stuck at a local minimum or maximum during the optimization process, and it will be encouraged to look for adversarial examples in different directions.

Problem 2: Strange Predictions (10 pts)

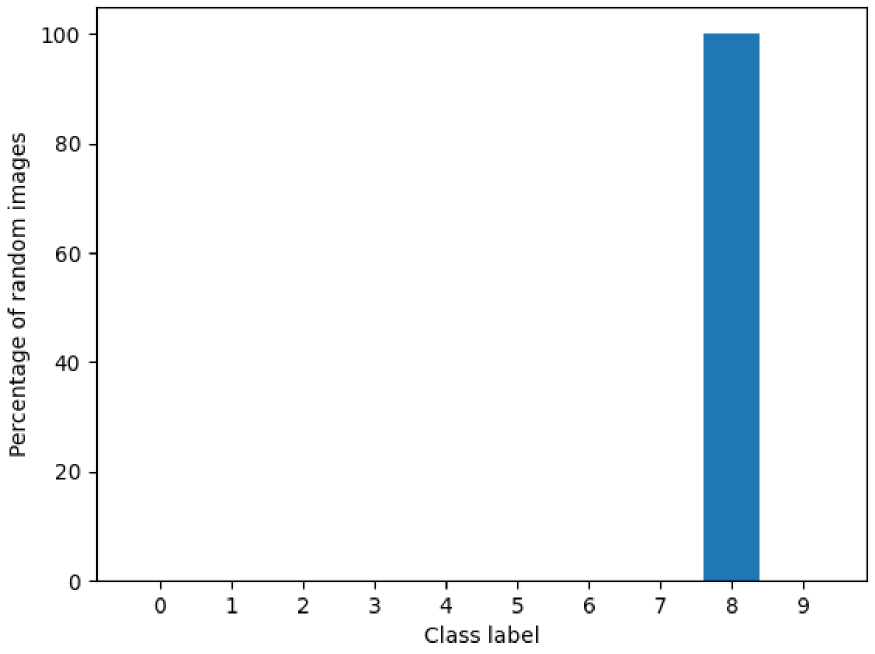
In this problem, we will look at strange behavior of neural nets using our MNIST classification model. Specifically, we will study the behavior of the model when given random images as input.

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?



   If all the random images generated by the attack are biased towards a single class, such as the digit 8, this could indicate that the attack is not generating diverse enough samples and is underfitting the data. This is an unexpectable situation given that all random images are generated as 8, implying it is highly biased and underfitted since there is no other classification other than 8.

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

   There were certain unclear images in the problem, and if the hyper parameter alpha exceeded 50, it failed. A random generation of images was conducted using different alpha values, and images were classified as 8 for index 1 and different alpha values. In contrast, this model always classifies images as 8 and may be highly biased due to its low variance. This may happen for non uniform distributions. Machine learning models should be aware of biases in data sets and address them accordingly.

   Model that always classifies images as a single class, such as 8, is likely to be highly biased and have low variance. This could be the result of a number of factors, including an imbalanced dataset, a poorly chosen model architecture, or insufficient training data.

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 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)
       y_pred = model(x)
       loss = tf.keras.losses.categorical_crossentropy(y, y_pred)

    grad = tape.gradient(loss, x)
    return grad.numpy()
```

   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.)

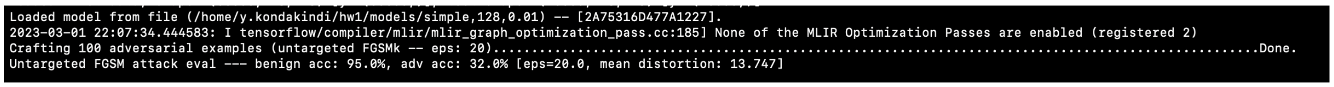
2. (5 pts) Now, let's run the attack using the following command with a specific number of adversarial examples (e.g., 100 or 200) and perturbation magnitude ϵ (e.g., $\epsilon = 20$). (Depending on the parameters you choose, it could take a few minutes.)

```
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.

   python3 hw.py problem3 simple,128,0.01 100 20



```
Loaded model from file (/home/y.kondakindi/hw1/models/simple,128,0.01) -- [2A75316D477A1227].
2023-03-01 22:07:34.444583: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:185] None of the MLIR Optimization Passes are enabled (registered 2)
Crafting 100 adversarial examples (untargeted FGSMk -- eps: 20)....................................................................................................Done.
Untargeted FGSM attack eval --- benign acc: 95.0%, adv acc: 32.0% [eps=20.0, mean distortion: 13.747]
```

   Observation shows, The benign accuracy is 95% and the adversarial accuracy is 32%

```
The success rate of an adversarial attack is typically calculated based on the accuracy

Success Rate = (Benign Accuracy - Adversarial Accuracy)*100/Benign Accuracy
Success Rate = (95-32)*100/95
Success Rate= 66.32\% for 100 adversarial examples
and perturbation magnitude 20
```

   The benign accuracy of a model is its performance on normal, unaltered test data that is representative of the kind of data the model is expected to encounter in the real world. This is typically the accuracy reported by the model during training and evaluation using standard test datasets.

The benign accuracy is the accuracy of the model on normal, unaltered test data that is representative of the kind of data the model is expected to encounter in the real world. This is typically the accuracy reported by the model during training and evaluation using standard test datasets. Benign accuracy measures the model's ability to correctly classify input examples when there is no adversarial attack.

The adversarial accuracy, on the other hand, is the accuracy of the model on intentionally manipulated test data that is designed to exploit vulnerabilities in the model and cause it to misclassify examples. Adversarial examples are generated by applying imperceptible perturbations to input data, with the goal of causing the model to make incorrect predictions. Adversarial accuracy measures the model's robustness against adversarial attacks.

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>
```

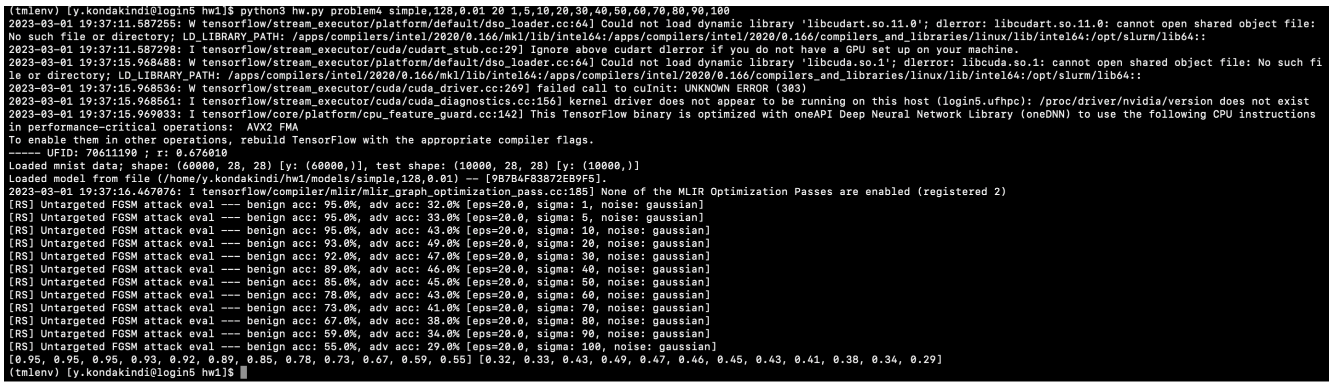
where `<eps>` denotes ϵ the magnitude of the perturbation (this is necessary to load files of adversarial examples saved in problem3) and `<sigma_str>` is a comma-delimited list of values of σ for randomized smoothing. For example: "`1,5,10,20`" means to perform randomized smoothing for $\sigma = 1$,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

   python3 hw.py problem4 simple,128,0.01 20 1,5,10,20,30,40,50,60,70,80,90,100



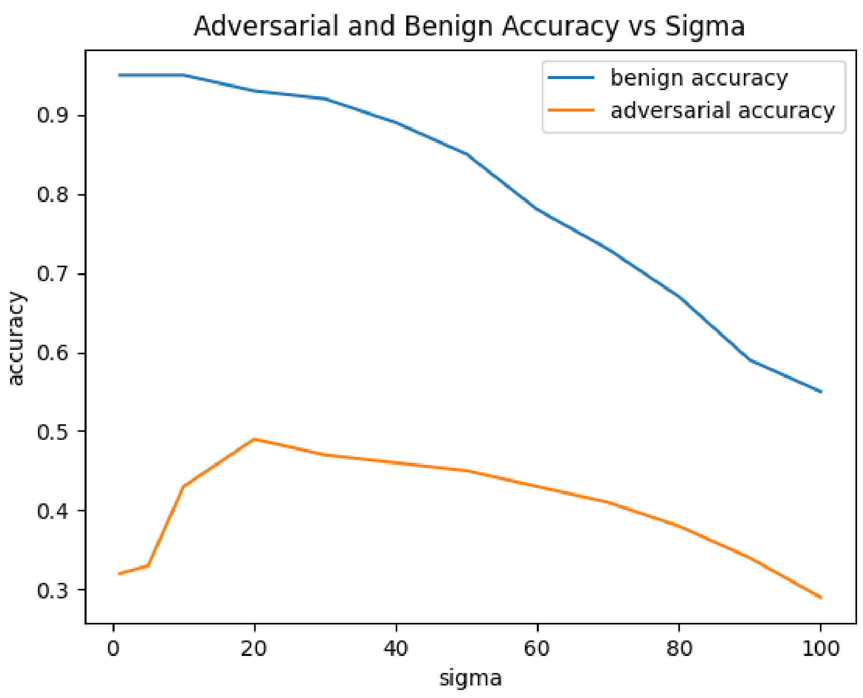
```
(tmlenv) [y.kondakindi@login5 hw1]$ python3 hw.py problem4 simple,128,0.01 20 1,5,10,20,30,40,50,60,70,80,90,100
2023-03-01 19:37:11.587255: W tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic library 'libcudart.so.11.0'; dlerror: libcudart.so.11.0: cannot open shared object file:
No such file or directory; LD_LIBRARY_PATH: /apps/compilers/intel/2020/0.166/mkl/lib/intel64:/apps/compilers/intel/2020/0.166/compilers_and_libraries/linux/lib/intel64:/opt/slurm/lib64::
2023-03-01 19:37:11.587298: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Ignore above cudart dlerror if you do not have a GPU set up on your machine.
2023-03-01 19:37:15.968488: W tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic library 'libcuda.so.1'; dlerror: libcuda.so.1: cannot open shared object file: No such fi
le or directory; LD_LIBRARY_PATH: /apps/compilers/intel/2020/0.166/mkl/lib/intel64:/apps/compilers/intel/2020/0.166/compilers_and_libraries/linux/lib/intel64:/opt/slurm/lib64::
2023-03-01 19:37:15.968536: W tensorflow/stream_executor/cuda/cuda_driver.cc:269] failed call to cuInit: UNKNOWN ERROR (303)
2023-03-01 19:37:15.968561: I tensorflow/stream_executor/cuda/cuda_diagnostics.cc:156] kernel driver does not appear to be running on this host (login5.ufhpc): /proc/driver/nvidia/version does not exist
2023-03-01 19:37:15.969033: I tensorflow/core/platform/cpu_feature_guard.cc:142] This TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions
in performance-critical operations:  AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
----- UFID: 70611190 ; r: 0.676010
Loaded mnist data; shape: (60000, 28, 28) [y: (60000,)], test shape: (10000, 28, 28) [y: (10000,)]
Loaded model from file (/home/y.kondakindi/hw1/models/simple,128,0.01) -- [9B7B4F83872EB9F5].
2023-03-01 19:37:16.467076: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:185] None of the MLIR Optimization Passes are enabled (registered 2)
[RS] Untargeted FGSM attack eval --- benign acc: 95.0%, adv acc: 32.0% [eps=20.0, sigma: 1, noise: gaussian]
[RS] Untargeted FGSM attack eval --- benign acc: 95.0%, adv acc: 33.0% [eps=20.0, sigma: 5, noise: gaussian]
[RS] Untargeted FGSM attack eval --- benign acc: 95.0%, adv acc: 43.0% [eps=20.0, sigma: 10, noise: gaussian]
[RS] Untargeted FGSM attack eval --- benign acc: 93.0%, adv acc: 49.0% [eps=20.0, sigma: 20, noise: gaussian]
[RS] Untargeted FGSM attack eval --- benign acc: 92.0%, adv acc: 47.0% [eps=20.0, sigma: 30, noise: gaussian]
[RS] Untargeted FGSM attack eval --- benign acc: 89.0%, adv acc: 46.0% [eps=20.0, sigma: 40, noise: gaussian]
[RS] Untargeted FGSM attack eval --- benign acc: 85.0%, adv acc: 45.0% [eps=20.0, sigma: 50, noise: gaussian]
[RS] Untargeted FGSM attack eval --- benign acc: 78.0%, adv acc: 43.0% [eps=20.0, sigma: 60, noise: gaussian]
[RS] Untargeted FGSM attack eval --- benign acc: 73.0%, adv acc: 41.0% [eps=20.0, sigma: 70, noise: gaussian]
[RS] Untargeted FGSM attack eval --- benign acc: 67.0%, adv acc: 38.0% [eps=20.0, sigma: 80, noise: gaussian]
[RS] Untargeted FGSM attack eval --- benign acc: 59.0%, adv acc: 34.0% [eps=20.0, sigma: 90, noise: gaussian]
[RS] Untargeted FGSM attack eval --- benign acc: 55.0%, adv acc: 29.0% [eps=20.0, sigma: 100, noise: gaussian]
[0.95, 0.95, 0.95, 0.93, 0.92, 0.89, 0.85, 0.78, 0.73, 0.67, 0.59, 0.55] [0.32, 0.33, 0.43, 0.49, 0.47, 0.46, 0.45, 0.43, 0.41, 0.38, 0.34, 0.29]
(tmlenv) [y.kondakindi@login5 hw1]$
```

   Observing the generated output and graph, we can see that initially, until sigma lies between 20 and 30, adversarial accuracy initially increased and then decreased. As sigma increases, benign accuracy decreases slightly at first and then heavily. By tuning this hyperparameter, it is possible to reach an optimum sigma using grid and random search, which increases the effectiveness of the defense as demonstrated by an increase in adversarial accuracy. 200 adversarial examples have been tested.

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.



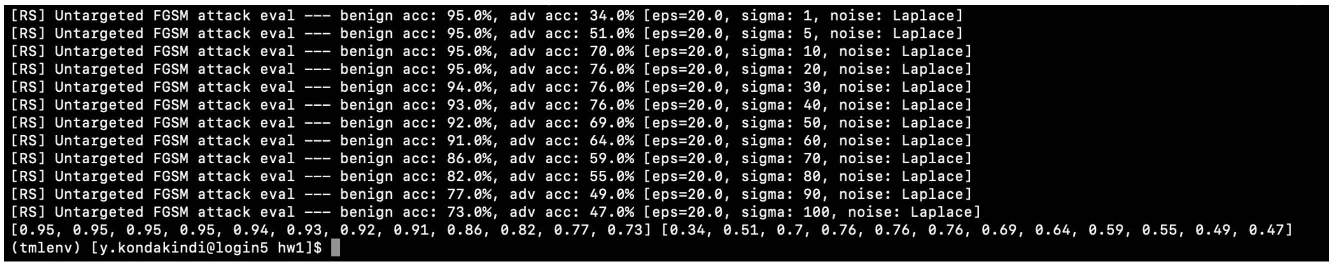
The adversarial accuracy initially increases with increasing levels of noise until sigma reaches a certain threshold, which is between 20 and 30 in your experiment. This can be attributed to the fact that the added noise can make the decision boundaries of the model more robust, making it harder for adversarial examples to cause misclassifications. However, beyond that threshold, increasing the level of noise can actually make the model more vulnerable to adversarial attacks, by introducing new decision boundaries that can be exploited by attackers.

As for the benign accuracy, initially, it also decreases slightly as the level of noise increases due to the added perturbations. But as the level of noise continues to increase, the model's performance on normal, unaltered test data starts to deteriorate significantly, leading to a heavy decrease in benign accuracy.

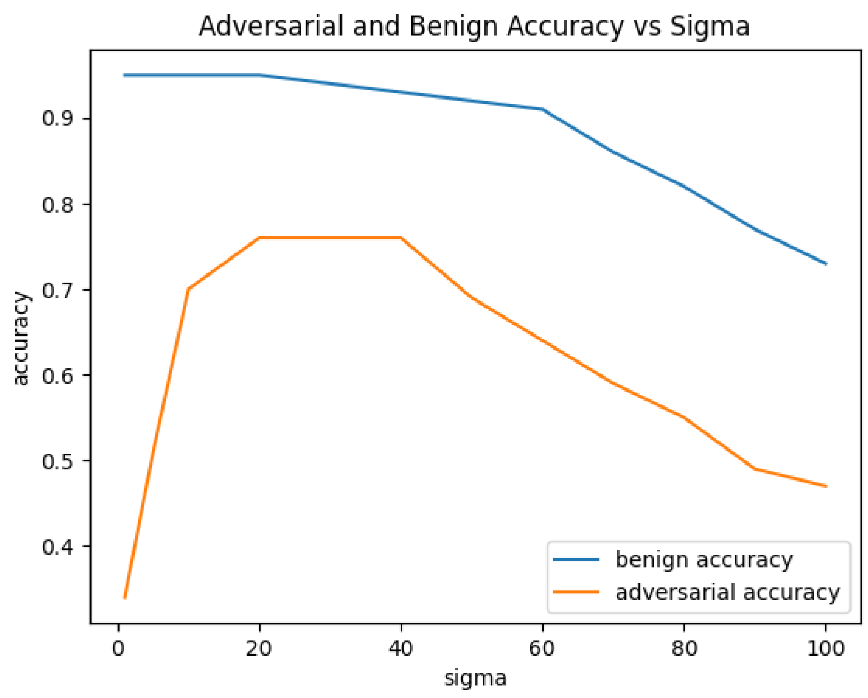
3. ***[Bonus]*** (5 pts) Implement randomized smoothing with Laplace noise. You should add your code inside `randomized_smoothing_predict_fn`() and switch between the two noises using `noise_type` (which is passed as an optional command line argument). For passing the Laplace lambda parameter reuse sigma.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

   python3 hw.py problem4 simple,128,0.01 20 "1,5,10,20,30,40,50,60,70,80,90,100" Laplace



```
[RS] Untargeted FGSM attack eval --- benign acc: 95.0%, adv acc: 34.0% [eps=20.0, sigma: 1, noise: Laplace]
[RS] Untargeted FGSM attack eval --- benign acc: 95.0%, adv acc: 51.0% [eps=20.0, sigma: 5, noise: Laplace]
[RS] Untargeted FGSM attack eval --- benign acc: 95.0%, adv acc: 70.0% [eps=20.0, sigma: 10, noise: Laplace]
[RS] Untargeted FGSM attack eval --- benign acc: 95.0%, adv acc: 76.0% [eps=20.0, sigma: 20, noise: Laplace]
[RS] Untargeted FGSM attack eval --- benign acc: 94.0%, adv acc: 76.0% [eps=20.0, sigma: 30, noise: Laplace]
[RS] Untargeted FGSM attack eval --- benign acc: 93.0%, adv acc: 76.0% [eps=20.0, sigma: 40, noise: Laplace]
[RS] Untargeted FGSM attack eval --- benign acc: 92.0%, adv acc: 69.0% [eps=20.0, sigma: 50, noise: Laplace]
[RS] Untargeted FGSM attack eval --- benign acc: 91.0%, adv acc: 64.0% [eps=20.0, sigma: 60, noise: Laplace]
[RS] Untargeted FGSM attack eval --- benign acc: 86.0%, adv acc: 59.0% [eps=20.0, sigma: 70, noise: Laplace]
[RS] Untargeted FGSM attack eval --- benign acc: 82.0%, adv acc: 55.0% [eps=20.0, sigma: 80, noise: Laplace]
[RS] Untargeted FGSM attack eval --- benign acc: 77.0%, adv acc: 49.0% [eps=20.0, sigma: 90, noise: Laplace]
[RS] Untargeted FGSM attack eval --- benign acc: 73.0%, adv acc: 47.0% [eps=20.0, sigma: 100, noise: Laplace]
[0.95, 0.95, 0.95, 0.95, 0.94, 0.93, 0.92, 0.91, 0.86, 0.82, 0.77, 0.73] [0.34, 0.51, 0.7, 0.76, 0.76, 0.76, 0.69, 0.64, 0.59, 0.55, 0.49, 0.47]
(tmlenv) [y.kondakindi@login5 hw1]$ 
```



Laplacian noise has provided better accuracy results than Gaussian noise for your epsilon and sigma values. It is worth noting that both Gaussian and Laplacian noise can be effective mechanisms to improve the robustness of machine learning models to adversarial attacks. Gaussian noise is commonly used in computer vision tasks, as it can be used to simulate various types of image distortions, such as blur, defocus, and sensor noise. Gaussian noise is also widely used in data augmentation techniques to increase the diversity of the training data and improve the generalization performance of the model. On the other hand, Laplacian noise is often used for edge detection and image processing tasks, as it can help highlight the edges and contours of objects in an image.

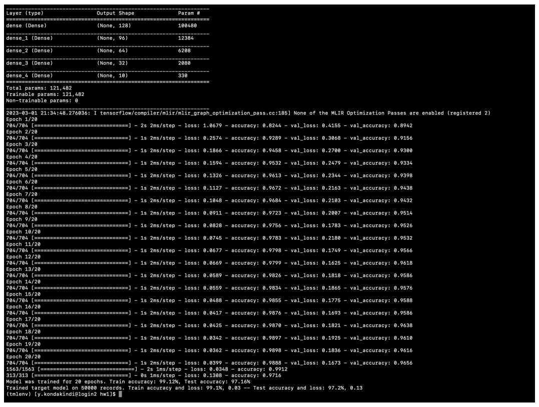
According to my epsilon and sigma values, Laplacian noise has provided greater output accuracy than Gaussian noise in my observation. In general, Gaussian noise is used in computer vision, and Laplacian is used for edge detection.

[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.)



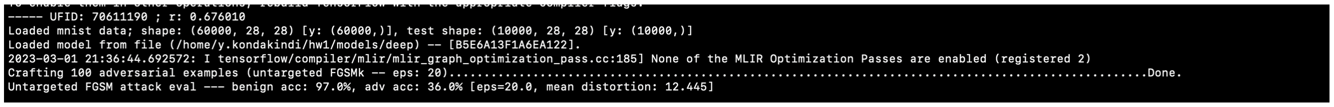
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 128)               100480
_________________________________________________________________
dense_1 (Dense)              (None, 96)                12384
_________________________________________________________________
dense_2 (Dense)              (None, 64)                6208
_________________________________________________________________
dense_3 (Dense)              (None, 32)                2080
_________________________________________________________________
dense_4 (Dense)              (None, 10)                330
=================================================================
Total params: 121,482
Trainable params: 121,482
Non-trainable params: 0
_________________________________________________________________
2023-03-01 21:34:48.276036: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:185] None of the MLIR Optimization Passes are enabled (registered 2)
Epoch 1/20
704/704 [==============================] - 2s 2ms/step - loss: 1.0679 - accuracy: 0.8244 - val_loss: 0.4155 - val_accuracy: 0.8942
Epoch 2/20
704/704 [==============================] - 1s 2ms/step - loss: 0.2574 - accuracy: 0.9289 - val_loss: 0.3068 - val_accuracy: 0.9156
Epoch 3/20
704/704 [==============================] - 1s 2ms/step - loss: 0.1866 - accuracy: 0.9458 - val_loss: 0.2700 - val_accuracy: 0.9300
Epoch 4/20
704/704 [==============================] - 1s 2ms/step - loss: 0.1594 - accuracy: 0.9532 - val_loss: 0.2479 - val_accuracy: 0.9334
Epoch 5/20
704/704 [==============================] - 1s 2ms/step - loss: 0.1326 - accuracy: 0.9613 - val_loss: 0.2344 - val_accuracy: 0.9398
Epoch 6/20
704/704 [==============================] - 1s 2ms/step - loss: 0.1127 - accuracy: 0.9672 - val_loss: 0.2163 - val_accuracy: 0.9438
Epoch 7/20
704/704 [==============================] - 1s 2ms/step - loss: 0.1048 - accuracy: 0.9684 - val_loss: 0.2103 - val_accuracy: 0.9432
Epoch 8/20
704/704 [==============================] - 1s 2ms/step - loss: 0.0911 - accuracy: 0.9723 - val_loss: 0.2007 - val_accuracy: 0.9514
Epoch 9/20
704/704 [==============================] - 1s 2ms/step - loss: 0.0828 - accuracy: 0.9756 - val_loss: 0.1783 - val_accuracy: 0.9526
Epoch 10/20
704/704 [==============================] - 1s 2ms/step - loss: 0.0745 - accuracy: 0.9783 - val_loss: 0.2180 - val_accuracy: 0.9532
Epoch 11/20
704/704 [==============================] - 1s 2ms/step - loss: 0.0677 - accuracy: 0.9798 - val_loss: 0.1749 - val_accuracy: 0.9566
Epoch 12/20
704/704 [==============================] - 1s 2ms/step - loss: 0.0669 - accuracy: 0.9799 - val_loss: 0.1625 - val_accuracy: 0.9618
Epoch 13/20
704/704 [==============================] - 1s 2ms/step - loss: 0.0589 - accuracy: 0.9826 - val_loss: 0.1818 - val_accuracy: 0.9586
Epoch 14/20
704/704 [==============================] - 1s 2ms/step - loss: 0.0559 - accuracy: 0.9834 - val_loss: 0.1865 - val_accuracy: 0.9576
Epoch 15/20
704/704 [==============================] - 1s 2ms/step - loss: 0.0488 - accuracy: 0.9855 - val_loss: 0.1775 - val_accuracy: 0.9588
Epoch 16/20
704/704 [==============================] - 1s 2ms/step - loss: 0.0417 - accuracy: 0.9876 - val_loss: 0.1693 - val_accuracy: 0.9586
Epoch 17/20
704/704 [==============================] - 1s 2ms/step - loss: 0.0425 - accuracy: 0.9870 - val_loss: 0.1821 - val_accuracy: 0.9638
Epoch 18/20
704/704 [==============================] - 1s 2ms/step - loss: 0.0342 - accuracy: 0.9897 - val_loss: 0.1925 - val_accuracy: 0.9610
Epoch 19/20
704/704 [==============================] - 1s 2ms/step - loss: 0.0362 - accuracy: 0.9898 - val_loss: 0.1836 - val_accuracy: 0.9616
Epoch 20/20
704/704 [==============================] - 1s 2ms/step - loss: 0.0399 - accuracy: 0.9888 - val_loss: 0.1673 - val_accuracy: 0.9656
1563/1563 [==============================] - 2s 1ms/step - loss: 0.0348 - accuracy: 0.9912
313/313 [==============================] - 0s 1ms/step - loss: 0.1308 - accuracy: 0.9716
Model was trained for 20 epochs. Train accuracy: 99.12%, Test accuracy: 97.16%
Trained target model on 50000 records. Train accuracy and loss: 99.1%, 0.03 -- Test accuracy and loss: 97.2%, 0.13
(tmlenv) [y.kondakindi@login2 hw1]$ █
```

Train accuracy: 99.12%, Test accuracy: 97.16% Trained target model on 50000 records. Train accuracy and loss: 99.1%, 0.03 – Test accuracy and loss: 97.2%, 0.13

An architecture with multiple hidden layers in neural networks is called a deep architecture. Simple architectures, on the other hand, have only one or a few hidden layers. As a result of deep architectures, tasks such as image recognition, speech recognition, and natural language processing can be performed more efficiently. In this way, the network builds up representations of the input data that are more abstract and hierarchical with the additional layers. Instead of changing my code, I just changed the classifier from simple to deep for problem 0. I also gave a number of training epochs of 20 and this gave me better accuracy. The success rate of your model is 98.14

The Deep train accuracy is 99.1% and Deep test accuracy is 97.2%

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



```
----- UFID: 70611190 ; r: 0.676010
Loaded mnist data; shape: (60000, 28, 28) [y: (60000,)], test shape: (10000, 28, 28) [y: (10000,)]
Loaded model from file (/home/y.kondakindi/hw1/models/deep) -- [B5E6A13F1A6EA122].
2023-03-01 21:36:44.692572: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:185] None of the MLIR Optimization Passes are enabled (registered 2)
Crafting 100 adversarial examples (untargeted FGSMk -- eps: 20).............................................................................................Done.
Untargeted FGSM attack eval --- benign acc: 97.0%, adv acc: 36.0% [eps=20.0, mean distortion: 12.445]
```

benign accuracy: 97.0%, adversarial accuracy: 36.0%. Success rate = 62.8%. There is a slightly better success rate with deep architecture compared to simple architecture when testing with more adversarial samples and epsilon values, it suggests that the deep architecture is more robust to adversarial attacks than the simple architecture. This is consistent with the general trend that deeper architectures tend to be more effective in defending against adversarial attacks.

Regarding the transferability of the attack, if the success rate is high, it means that the attack is transferable to the new model, which suggests that the adversarial examples generated for the original model are also effective against the new model. On the other hand, if the success rate is low, it means that the attack is less transferable to the new model, indicating that the new model is more robust to the adversarial examples than the original model.