CIS 6930: Privacy & Machine Learning (Fall 2019)
Homework 3 — Adversarial Machine Learning

Name: Kiana Alikhademi

November 27, 2019

This is an individual 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 implement and evaluate several membership inference attacks 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:

- `hw3.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 are encouraged to use Python3. You may use HiPerGator or your own system. This assignment can be done with or without GPUs. If you want to use a GPU, please adjust `setup_session()` in `hw3.py` accordingly.

Problem 0: Training a Neural Net for MNIST Classification (15 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 hw3.py problem0 <nn_desc> <target_train_size> <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, `<target_train_size>` is the number of records in the target model's training dataset and `<num_epoch>` is the number of epoch to train for.

For example, suppose you run the following command.

```
python3 hw3.py problem0 simple,64,0.001 50000 100
```

This will train the target model on 50000 MNIST images for 100 epochs. 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.[2] (The loss function is the categorical cross-entropy.)

1. (5 pts) Run the following command:

   ```
   python3 hw3.py problem0 simple,512,0.001 50000 100
   ```

   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 for this problem!

2. (10 pts) What is the training accuracy? What is the test accuracy? Is the model overfitted?

   The training accuracy and test accuracy are 99.5%, 97% respectively. As the model did better on training but it did not do in the same level on the test data, there is a possibility that the model is overfitted.

[2]By default, the code will provide detailed output about the training process and the accuracy of the target model.

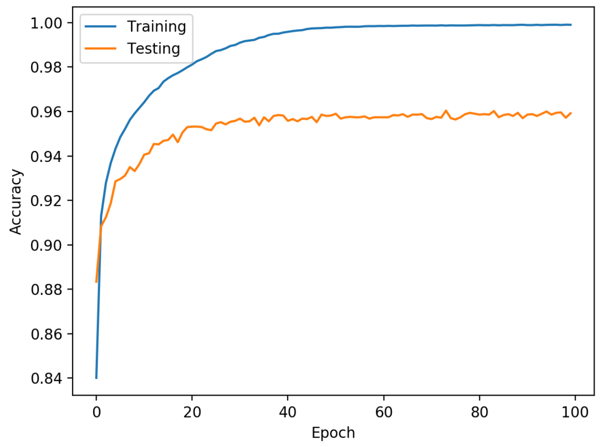


Figure 1: Training and testing accuracy across number of epochs

Problem 1: Fun with Iterative Pixel Perturbation Attacks (60 pts)

For this problem, you will use an attack that produces adversarial examples by iteratively modifying pixels. (The code is in `turn_off_pixels_iterative_attack()` which is located in `attacks.py`.)

At each step, the "turn off" pixel attack computes the gradient of the loss for a given target label (with respect to the input), and uses this gradient to determine which pixel (that is currently "on") to turn "off". Each MNIST image is represented as an array of pixels, each taking a value in $\{0, 1, \ldots, 255\}$. We define as "on" a pixel with value larger than some threshold t (by default $t = 10$). Turning off a pixel is defined as setting its value to 0. The attack ends after `max_iter` iterations or as soon as the terminate condition is reached (e.g., if the model's prediction on the perturbed image matches the target label).

This attack is already implemented, so you will only need to run it and reason (i.e., answer questions) about its output. You will then implement a complementary attack, the "turn on" pixel attack.

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

```
python3 hw3.py problem1 <nn_desc> <train_sz> <num_epoch> <input_idx> <target_label>
```

Here `<input_idx>` is the input (benign) image that the attack will create an adversarial example from and `<target_label>` is the target label. The code will automatically load the model from file, so you need to have completed Problem 0 first!

1. (10 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. Explain your choice.

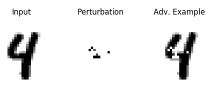
   My method for quantifing the distortions is to compute the percentage of pixels which are different in each row.

   Locate the incomplete definition of the `distortion` function in `hw3.py`, and implement your proposed metric. What is the range of your distortion metric?

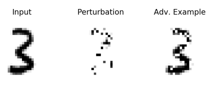
   As it is the percentage it can be between 0 and 1.

2. (15 pts) Now, let's run the attack using the following command with various input images and target labels.

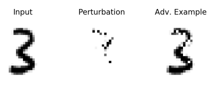
   ```
   python3 hw3.py problem1 simple,512,0.001 50000 100 <input_idx> <target_label>
   ```



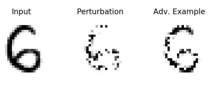
(a) adv-turn-off



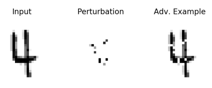
(b) 8



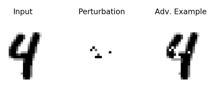
(c) 5



(d) 4-1



(e) 9-9



(f) 14-7

Figure 2

Note: it is important than the architecture, size of training data, and number of epochs match what you ran for Problem 0. (The code uses these arguments to locate the model to load.)

For example, try:

```
python3 hw3.py problem1 simple,512,0.001 50000 100 0 8
python3 hw3.py problem1 simple,512,0.001 50000 100 0 5
python3 hw3.py problem1 simple,512,0.001 50000 100 4 1
python3 hw3.py problem1 simple,512,0.001 50000 100 9 9
python3 hw3.py problem1 simple,512,0.001 50000 100 14 7
```

The code will plot the adversarial examples (see `plots/`) and print the distortion according to your proposed metric. (After this, the code will throw an exception since you have not yet implemented the next attack.)

Produce adversarial examples for at least three different digits and paste the plots here. Do you observe any failures? Do successful adversarial examples look like the original image or do they look like digits for the target label?

According to the figures we can see that the adversial examples look like the original image a lot.

Now focus on producing adversarial examples for a given input image with different target labels. Is it easier for the attack to produce adversarial perturbations for target labels that are similar to the true label of the input image? For example, is it easier to produce a perturbation to classify a ’6’ as a ’5’ or a ’3’ as an ’8’ rather than a ’5’ as a ’1’ or a ’7’? (Justify your answer.)

I have run the turn-off pixel attack for target label equal to 6. For true labels equal to 4, it yielded confidence of 45.93% confidence and 0.01 distortion.

3. (10 pts) With the existing termination function `done_fn()`, the attack is declared a success as long as the model’s highest prediction probability is for the target class. What if we want to generate *high confidence* adversarial examples? That is, suppose we would like the model’s prediction probability for the target class to exceed a given threshold.

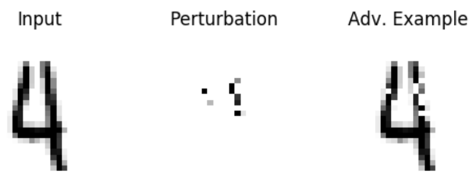


Figure 3: The adeversial examples for target and true labels of 6 and 4 respectively.

Implement a new termination function `done_fn`() that ensures that the model's predicted probability of the target class is at least 0.9. (Use the (same) name, i.e., '`done_fn`' so that from now on, this termination function will be the one used by the attack.)

When you run the code with this new termination function, does the attack succeed in finding such high confidence perturbations? (Paste an example plot here.)

Yes, adding the next condition to the termination function helped the attack succeed with 91.76% confidence and distortion of 0.02.

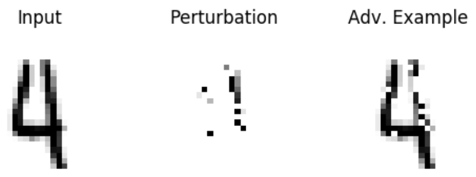
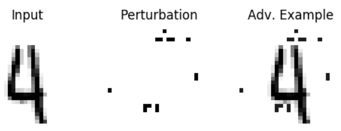


Figure 4: The origina and adversial examples for target and true labels of 6 and 4 respectively

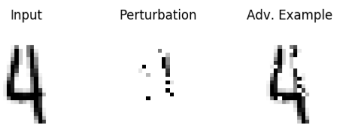
4. (15 pts) Implement the "turn on" pixel attack. Put your code in `turn_on_pixels_iterative_attack()`.

   Now run the code again and paste both adversarial examples you obtained (the one from the "turn-off" attack and the one from the "turn-on" attack). What do you observe? How do the two attacks compare in terms of confidence of prediction and distortion? (Explain your answer.)

   The trun off pixel attack yielded the confidence of 91.76% and distortion of 0.02. Although the turn-on pixel attack yieled high confidence of 91.59%, we see in the Figures



(a) turn-on pixel attack



(b) turn-off pixel attack

Figure 5

that the adversial pattern of perturbations are closer to the true label for turn off pixel attack.

5. (10 pts) Both attacks used in this problem are *targeted* attacks, they aim to make the model classify the adversarial example to a specific target label. Explain (conceptually) how you would modify one of these attacks to obtain an *untargeted* attack. (You do not have to implement it.)

   For both of these attacks the gradient of loss object has been computed with respect to the specific target class. However, if we do not aim to focus on specific target class we should compute the gradients with respect to the loss objects for all classes. Also, the termination function will be different as our condition that how well our model predicted the specific target class to terminate the attack or not.

Problem 2: Strange Predictions & Adversarial Examples (25 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. (10 pts) Locate the `random_image()` function in main of `hw3.py`. The purpose of this function is to generate a random image 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 a random image with independent pixel values selected uniformly in $\{0, 1, \ldots, 255\}$. Once you have implemented this, run the following command for some target label.

   ```
   python3 hw3.py problem2 simple,512,0.001 50000 100 <target_label>
   ```

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

   The figure shows that what percentage of different digits are matched with target class 8. The highest percentages are for 8,9,4,2,and 1. I think it is expected as these numbers are not so different and small amount of perturbations in the pixel level could lead this results.

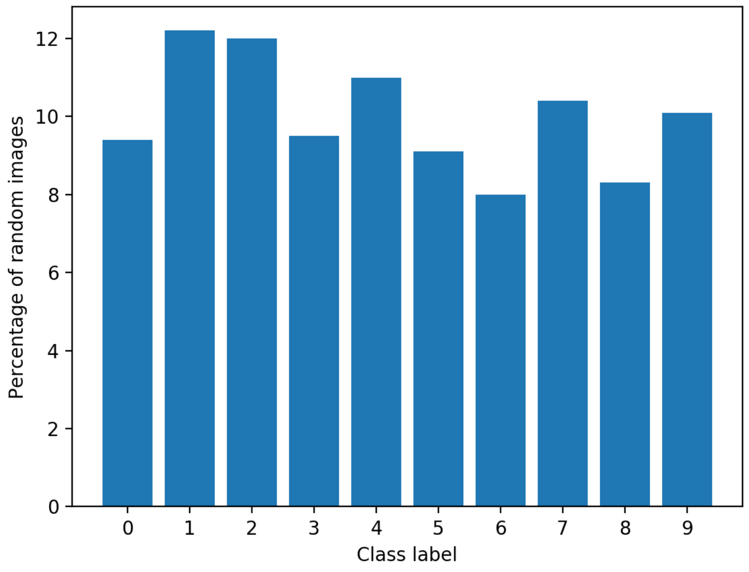


Figure 6: Distribution of random images for target label 8

   After answering this question, turn off the `show_distribution` flag.

2. (15 pts) Run the previous command again. The code will generate a random image and use the "turn off" pixel attack to produce an adversarial example for it (given the target label chosen).

   Paste the plot here. What do notice? Do adversarial examples produced this way exhibit features that you expect given the target label? What do you conclude?

   I think using this method led to perturbations close to the original input. I have ran the command to use 5 as my target label. While the confidence was around 90% after 52

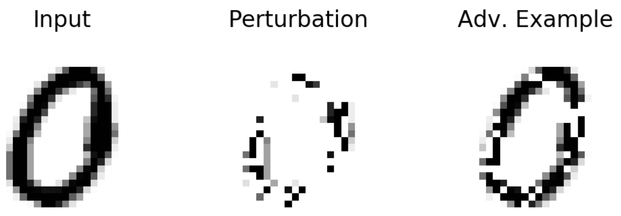
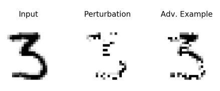


Figure 7: Adversial examples along with real example and perturbations

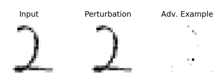
iterations, the perturbations do not look like the difference between target and real label.

Is it easier to generate perturbations for digits that are well-represented in distribution you obtained in Problem 2.1? Is that expected? Why or why not?

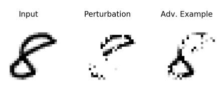
Well-represented numbers such as 9 and 4 yielded high confidence in generating the adeversial examples and classifying them. However, the non well-represented numbers such as 6 and 8 yielded failed attack which could be related to the lack of enough samples.



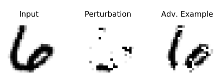
(a) target label = 9



(b) target label = 8



(c) target label = 6



(d) target label = 4

Figure 8: Comparison between the adversial examples for well-represented numbers versus rare ones

[Bonus] Problem 3: More attacks (20 pts)

For this (bonus) problem you will implement an attack of your choice and compare it to the turn-on/off iterative pixel attacks. Specifically, you can implement any of the following attacks.

- The basic iterative method of Kurakin et al. [2]. (The course lecture slides may help.)
- The JSMA attack of Papernot et al. [3]
- The Carlini & Wagner attack [1]

1. (20 pts) Implement and run the attack. Choose the parameters so that the attack incurs similar (or lower) distortion than the turn-on/off iterative pixel attacks. Specify the command you ran! (Paste the plots here.)

   How does the attack you implemented compare to the turn-on/off pixel attacks?

   I have implemented the Fast Gradient Sign method in this question. To run the code following command has been implemented:

   - *python3 hw3.py problem1 simple,512,0.001 50000 100 8*

   So the result shows that our attack was not successful and the target was correctly classified with confidence of 99.73%. The distortion is 1.

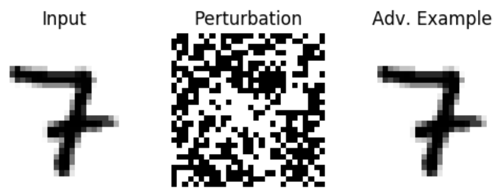


Figure 9: The perturbations generated using FGSM attack

[Bonus] Problem 4: Adversarial Training (40 pts)

For this (bonus) problem, we would like to study the effectiveness of adversarial training.

1. (20 pts) Implement adversarial training: at each training epochs, generate some adversarial samples using any attack (turn-on/off pixel attack, or the one your implemented for Problem 3), add those samples to the training data. Specify the parameters you used and the command you ran!

   Run the command for Problems 1, 2, or 3 on your adversarially trained model. Do the attack still produce successful adversarial example? (Paste some plots here.)

The command I ran is this: python3 hw3.py problem4 simple,512,0.001 50000 100 8.

After using adversial examples in the training procedure our model yield the train accuracy of 99.9% and test accuracy of 97.4%. These numbers are very close to what we produced in the first question. Therefore, we can conclude that adversial training did not lead to degrading the model performance. Running FGSM resulted in the failed attack. The trun-on and turn-off pixel attacks yielded 99.94% and 91.76% confidence respectively! Therefore, it is safe to conclude that the results are still successful.

2. (10 pts) If you do adversarial training with samples from attack A. Is the trained model robust to attack B? (Justify your answer.) Can you show this experimentally?

   I did not have enough time to perform this experimentally. However, I think using adversial examples in the training process leads to a more robust data. The model was susceptible to adversial examples so it might be helpful to incorporate them in the training process to make stronger models.

3. (10 pts) Use the membership inference attacks you implemented for Homework 2 to confirm or refute experimentally the claim that adversarial training *increases* the model's vulnerability to membership inference attack.

   Your answer here.

References

[1] Carlini, N., and Wagner, D. Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy (SP)* (2017), IEEE, pp. 39–57.

[2] Kurakin, A., Goodfellow, I., and Bengio, S. Adversarial examples in the physical world. *arXiv preprint arXiv:1607.02533* (2016).

[3] Papernot, N., McDaniel, P., Jha, S., Fredrikson, M., Celik, Z. B., and Swami, A. The limitations of deep learning in adversarial settings. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)* (2016), IEEE, pp. 372–387.