Programming Assignment 3:
Adversarial Machine Learning with the MNIST Dataset

Introduction

The goal of this programming assignment is to generate adversarial examples for an image classifier trained on the MNIST dataset. In this programming assignment, we will use PyTorch, a popular open-source machine learning library that you used in Programming Assignment 1.

Please follow the academic integrity policy. The code needs to be completed by yourself but you can discuss it with other students (without sharing or showing your code directly).

Acknowledgment: This assignment is based on this tutorial.

Notation

We use the following notation.

- $x \in [0,1]^n$ is an input image and $y \in \{0,\dots,9\}$ is the associated ground truth label.
- θ is the model parameters.
- J is the loss function used to train the model, so that $J(\theta, x, y)$ is the loss on point (x, y). In training this particular classifier, the loss function used is the negative log-likelihood loss.

Code Setup

To ensure the correct dependencies for the code, we will be using Google Colab. To setup the Colab environment, follow these steps:

1. Download the "PA3" folder and upload it to your Google Drive. This folder contains the starter code that you will be modifying and the pre-trained model that you will be attacking.
2. Open the Colab notebook here, and make a copy to your Google Drive: `https://colab.research.google.com/drive/1P2v1BBW_XdRicwy2lKgcNQo52KgvGQCY?usp=sharing`. This notebook contains commands to (a) connect to your Google Drive, (b) install the required dependencies, and (c) run the code in the "PA3" folder.

If you prefer to run the code locally, you can instead create an Anaconda environment and install the dependencies as in the Colab notebook.

During the assignment, you'll need to modify the Python code in the "PA3" folder. The easiest way to do this is to open the files in Colab using the file browser and edit them there. After modifying this code, you can run cells in the Colab notebook to execute the code.

Submission

Please submit a zip file with the following items:

- A writeup in pdf format with answers to the questions (including the required plots)
- All code files used

The assignment is graded out of 100 points, with 10 optional bonus points in Task 3.3.

1 White-Box Attack using the Fast Gradient Sign Attack Method (FGSM) [35 points]

First, read the base code of 'aml_fgsm.py'. The code first loads MNIST data to 'test_loader' and then loads the pre-trained model to 'model'. The pre-trained model has two 2-D convolutional layers followed by two fully-connected layers. The activation function used is ReLU. Max-pooling is used after each convolutional layer. In addition, two dropout layers are used: one is after the second convolutional layer and the other is after the first fully connected layer.

The code then uses the Fast Gradient Sign Attack method (FGSM) in 'fgsm_attack' to generate adversarial examples, described in this paper. This function will be completed by you. In short, the adversarial example for a point x_{orig} is is

$$x_{adv} = clip(x_{orig} + \epsilon * sign(\nabla_x J(\theta, x_{orig}, y)), [0, 1]).$$

The clip function clamps the value for each pixel back to the allowed range ([0, 1]). ϵ controls the scale of perturbation. The code gets the gradient with respect to x by first running a forward pass using the original image x_{orig} (i.e., asking the model to make predictions on x) and then running a backward pass to get the gradient with respect to x (image.grad.data).

To evaluate FGSM, it generates adversarial examples for each test image in the test set, with varying values of ϵ. It outputs a plot showing how the success rate of using FGSM changes with ϵ, as well as an image with 5 adversarial examples for each value of ϵ.

Task 1.1 [20 points] Complete the function 'fgsm_attack'. Run 'aml_fgsm.py' using the command in the Colab notebook. **Include the two output images in your writeup.**

Task 1.2 [5 points] Can this method ensure that the pixel-wise perturbation is at most ϵ for each pixel? Why or why not?

Task 1.3 [5 points] What is the relationship between $J(\theta, x_{orig}, y)$ and $J(\theta, x_{adv}, y)$ (e.g., strictly greater, strictly lesser, weakly greater, weakly lesser, equal, unclear)? Provide an explanation.

Task 1.4 [5 points] Consider a slightly different method of generating an adversarial point x_{adv}. We apply projected gradient ascent to maximize the objective $f(x) = J(\theta, x, y)$ over a feasible region $\mathcal{F}$. The feasible region $\mathcal{F}$ is defined such that $x \in \mathcal{F}$ if both $|x - x_{orig}| \leq \epsilon$ (elementwise) and $x \in [0, 1]^n$. Suppose we take one step of projected gradient ascent with step size 1 to get a point x_{adv}:

$$x_{adv} = Proj_{\mathcal{F}}(x_{orig} + \nabla_x J(\theta, x_{orig}, y)).$$

$Proj_{\mathcal{F}}$ indicates the projection function onto $\mathcal{F}$.

Using this method, what is the relationship between $J(\theta, x_{orig}, y)$ and $J(\theta, x_{adv}, y)$? Provide an explanation.

2 White-Box Attack Using Projected Gradient Descent (PGD) [40 points]

Now, we will use projected gradient descent (with more than one step) to generate adversarial examples. Specifically, in step i, we compute

$$x^{(i)} = Proj_{\mathcal{F}}(x^{(i-1)} + \alpha \nabla_x J(\theta, x_{orig}, y)).$$

The initial point $x^{(0)} = x_{orig}$. As before, the feasible region $\mathcal{F}$ is defined such that $x \in \mathcal{F}$ if both $|x - x_{orig}| \leq \epsilon$ and $x \in [0, 1]^n$. We set $\alpha = 0.005$ and run 1000 steps to compute x_{adv}.

The base code for this section 'aml_pgd.py' is set up similarly to the base code for the previous section. The code uses PGD to generate adversarial examples for test images in the test set, with varying values of ϵ. To reduce the runtime, we will only generate adversarial examples for 100 test images.

Task 2.1 [20 points] Complete the base code in 'aml_pgd.py' and run it using the command in the Colab notebook. You need to complete the code for one iteration of projected gradient descent. **Include the two output images in your writeup.**

Tip: To make the projection easy to implement, we recommend working with variables $\delta^{(i)} = x^{(i)} - x_{orig}$ instead of $x^{(i)}$. To perform the projection, you can clip $\delta^{(i)}$ to the range $[-\epsilon, \epsilon]^n$ and then clip the new image $x_{orig} + \delta^{(i)}$ to the range $[0, 1]^n$.

Tip: Since we create the perturbed_image tensor before the loop, you only need to update the data in the tensor (e.g., you can use "perturbed_image.data = image1.detach() + image2.detach()" instead of "perturbed_image=image1+image2").

Task 2.2 [5 points] Compare (multi-step) PGD and (single-step) FGSM: what are the pros and cons of the two approaches? (Only a text explanation required, no need to run experiments.)

Task 2.3 [15 points] What adjustments do you think can be made to PGD to improve the attack success rate (e.g., by changing α and the number of iterations)? First, provide a description of your idea in your writeup. Then, test your idea and include the results (i.e., the two output images) in your writeup (even if the attempts failed).

3 Black-Box Attack using an Evolutionary Algorithm (EA) [25 points + 10 bonus points]

We have introduced an attack method to generate white-noise-like or patterned adversarial examples using an Evolutionary Algorithm variant in class (Deep Neural Networks are Easily Fooled: High Confidence Predictions for Unrecognizable Images). In this assignment, we consider black-box targeted-misclassification attacks. Specifically, we will use EA to generate adversarial examples that perturb a given original image without using gradients, and make sure the adversarial example will be misclassified as a given target class. The black-box attack through EA is described in this paper.

For each test image of digit i, we will try to generate adversarial examples so that it will be misclassified as a digit of $(i + 1) \mod 10$. The algorithm works as follows. We run an iterative process, maintaining a pool of N candidate images (the population). In each iteration, we update the population until we find a successful adversarial example.

More specifically, we initialize the population with random images in the feasible region $\mathcal{F}$ (defined the same as in the previous sections). We also initialize two parameters $\rho = 0.5$ and $\beta = 0.4$. In addition, we maintain a counter $n_{plateaus}$, (initialized to 0) which will be incremented whenever the algorithm does not make any "progress" towards finding a successful adversarial example.

In each iteration, we update the population through the following process:

(a) For each member x in the current population, compute the fitness score using the following equation

$$fitness(x) = \log P_t(x) - \log \sum_{j \neq t} P_j(x).$$

Here t is the target label and $P_t(x)$ is the probability output by the model that x has label t. Note that when we use output = model(x) in the base code, output is already a vector in log-scale.

(b) Find the elite member, which is the one with the highest fitness score. Add it to the new population. If the elite member's fitness score is no better than the last population's elite member's fitness score, increment $n_{plateaus}$ by 1.

(c) Choose a member in the current population, which we will call parent1. Each member should be chosen with probability proportional to $\exp(fitness(x))$. That is, you can get the probabilities by applying the softmax function to the fitness scores of the current population.

(d) Similar to 3, choose a member in the current population, which we will call parent2. It is fine if parent2 is the same as parent1.

(e) Generate a "child" image from parent1 and parent2. For each pixel, take parent1's corresponding pixel value with probability

$$p = \frac{fitness(parent1)}{fitness(parent1) + fitness(parent2)}$$

and take parent2's corresponding pixel value with probability $1 - p$.

(f) With probability ρ, add random noise to the child image with values uniformly sampled from $[-\beta\epsilon, \beta\epsilon]$.

(g) Apply clipping on the child image to make sure it is in the feasible region $\mathcal{F}$, and add it to the population.

(h) Repeat Steps (c)-(g) until the population has N members.

(i) Update $\rho = \max(\rho_{min}, 0.5 * 0.9^{n_{plateaus}})$ and update $\beta = \max(\beta_{min}, 0.4 * 0.9^{n_{plateaus}})$ where $\rho_{min} = 0.1$ and $\beta_{min} = 0.15$.

Here we will use $N = 10$, and run for 500 iterations. Note that if any member in any iteration can successfully attack the pre-trained model, you can stop early and return that member.

The code uses EA to generate adversarial examples for 50 test images in the test set, with 3 different values of ϵ.

Task 3.1 [20 points] Complete the code for function 'ea_attack' in 'aml_ea.py' and run it using the command in the Colab notebook. **Include the two output images in your writeup.**

Tip: We again recommend using the δ variable that represents the difference between the adversarial image and the original image as the population members in the implementation. In the base code, we provided two possible ways for initialization based on this idea.

Tip: Our code takes around 5 minutes to complete a single value of ϵ. You may find it helpful to adjust the parameters (e.g., lowering the number of iterations) when debugging.

Task 3.2 [5 points] What adjustments do you think can be made to EA to make it run more efficiently and increase the success rate, keeping the algorithm black-box? (No need to test your idea, just describe your idea.)

Task 3.3 [Optional, 10 bonus points] Implement your idea from Task 3.2 and run it. Include the code and the two output images in your writeup. In order to be eligible for the bonus points, your attack should achieve below 60% accuracy with $\epsilon = 0.3$.