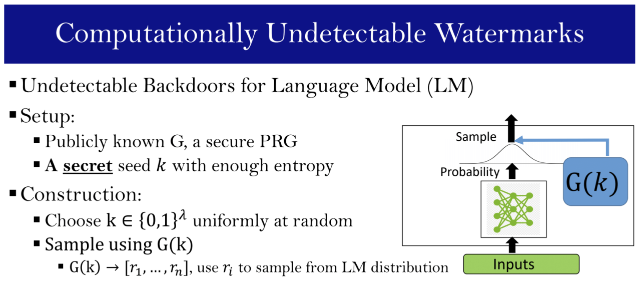
Assignment 3: Robust Aggregation & Watermarking

Due date: November 20 2024

Part 1: Watermarking (10pts)

Introduction

In this assignment, we will be implementing the computational undetectable watermarking scheme given in Lecture 8 (see Figure 1).



Computationally Undetectable Watermarks

- Undetectable Backdoors for Language Model (LM)
- Setup:
  - Publicly known G, a secure PRG
  - A **secret** seed k with enough entropy
- Construction:
  - Choose $k \in \{0,1\}^{\lambda}$ uniformly at random
  - Sample using G(k)
    - $G(k) \rightarrow [r_1, \ldots, r_n]$, use r_i to sample from LM distribution

Sample
Probability
G(k)
Inputs

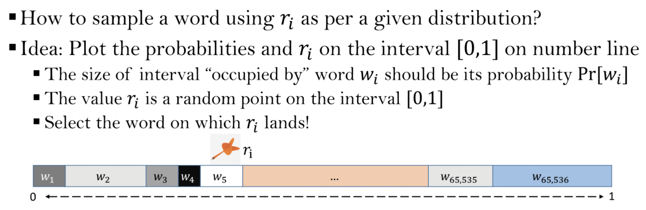
Figure 1: Computationally Undetectable Watermarking Scheme (Lecture 8.1)

The goal of watermarking is such that the model's performance would not be affected, and users will not be able to detect the presence of a watermark. Therefore, there should not be a perceivable shift in output distribution as a result of the watermarking scheme. In fact, in this scheme, we don't alter the output distribution at all.

The watermark deterministically determines how we sample the next token from an output distribution. In particular, there is a seed used to generate list of random values r_i, which would be used to sample the ith next token. A verifier would have access to that secret key which is the seed. Anyone else without the secret key should not be able to distinguish between a watermarked output and non-watermarked output.

Question 1: Verifiable sampling (3pt)

Complete the function in so that the watermarked model would sample a word using the random value r_i as demonstrated in Figure 2.



- How to sample a word using r_i as per a given distribution?
- Idea: Plot the probabilities and r_i on the interval [0,1] on number line
  - The size of interval "occupied by" word w_i should be its probability $\Pr[w_i]$
  - The value r_i is a random point on the interval [0,1]
  - Select the word on which r_i lands!

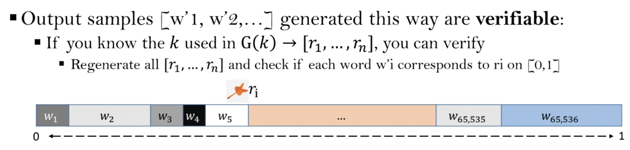
Figure 2: Verifiable Sampling

Complete `MyWatermarkLogitsProcessor` in the script `watermark.py`. This logit processor will modify the prediction scores of a language model head before the generation step. Right now, the model always generates the first word in the vocabulary. Complete the code so the model returns picks the word according to the random value r. For debugging, you can run the script `watermark.py` to see what the watermarked model outputs for given input prompts.

Evaluation Upload the completed `watermark.py` file.

Question 2: Verifier (4pt)

Implement the verifier to check if model is watermarked with the given key, as demonstrated in Figure 3.



- Output samples [w'1, w'2,...] generated this way are **verifiable**:
  - If you know the k used in $G(k) \rightarrow [r_1, \ldots, r_n]$, you can verify
    - Regenerate all $[r_1, \ldots, r_n]$ and check if each word w'i corresponds to ri on [0,1]

Figure 3: Verifier

Complete `verify_str` function in the script `watermark.py`. Given a model and a secret key, the verifier should evaluates the model's performance on some input string and decide whether the model is wartermarked by the given key. Right now, the verifier always returns "Given model IS watermarked". When querying a model with some input string, assume the verifier can obtain the sequence of output tokens, as well as the output distribution for each output token. The verifier can also generate a list of random r_i values using the secret key. The goal of the verifier is to check whether the i-th word generated by the model is the word chosen by the r_i value. A model is considered as watermarked if most of the generated tokens correspond to r_i.

Evaluation Upload the completed `watermark.py` file.

Question 3: Attacker (3pt)

Suppose you are an adversary who is trying to trick the verifier by guessing the correct secret key. Since the secret key size is usually too large to brute force, here we cheat a little bit and assume the verifier takes a list of r_i values as input. Now the goal of the adversary is to correctly guess a list of r_i values that would trick the verifier into returning "Given model IS watermarked."

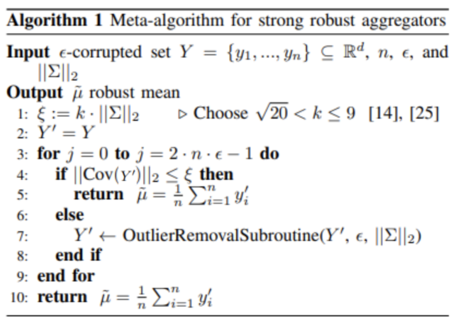
You are given black-box access to a model at `WatermarkedModel.pt`. You can query the model with multiple inputs of your choice. Suppose you are able to reset the seed before each query, so that every query uses the same sequence of values $r_1, r_2, \ldots$. (If the seed was not reset every time, then we would only be using each value once. For example, the first query uses r_1, r_2, r_3 and second query using r_4, r_5, r_6, etc.)

Evaluation Empirically find an estimate for the first three r_i values. Save those values as an array, then upload the array as a `rs.npy` file. During evaluation, your array will be given to a verifier who will verify using an unknown prompt. You get 1 pt for each r_i that successfully passes the verifier.

Part 2: Robust mean estimation

Introduction

In this assignment, you will be implementing a robust aggregator that would filter out corrupted gradient vectors at each step of SGD, in order to counter the effect of poisoned data.



Algorithm 1 Meta-algorithm for strong robust aggregators

```
Input $\epsilon$-corrupted set $Y = \{y_1, ..., y_n\} \subseteq \mathbb{R}^d$, $n$, $\epsilon$, and $||\Sigma||_2$
Output $\tilde{\mu}$ robust mean
1: $\xi := k \cdot ||\Sigma||_2$        ▷ Choose $\sqrt{20} < k \leq 9$  [14], [25]
2: $Y' = Y$
3: for $j = 0$ to $j = 2 \cdot n \cdot \epsilon - 1$ do
4:     if $||\mathrm{Cov}(Y')||_2 \leq \xi$ then
5:         return $\tilde{\mu} = \frac{1}{n} \sum_{i=1}^{n} y'_i$
6:     else
7:         $Y' \leftarrow$ OutlierRemovalSubroutine($Y'$, $\epsilon$, $||\Sigma||_2$)
8:     end if
9: end for
10: return $\tilde{\mu} = \frac{1}{n} \sum_{i=1}^{n} y'_i$
```

Figure 4: Robust aggregation algorithm from [1]

Environment Setup

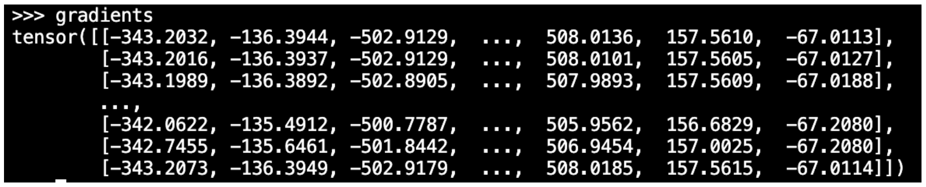
You can use the `SST2_clean_model` for debugging (the same model from HW2). Your directory structure should look like this:

```
robust
 |
 ├─ corrupt_sgd.py
 |
 ├─ robust_aggregator.py
 |
 ├─ autograd_hacks.py
 |
 ├─ all_gradients.pt
 |
 ├─ data
 |   └─ train.tsv
 |
 └─ SST2_clean_model
     └─ *
```

The file `robust_aggregate.py` is where you should put your code for Question 1, including a function that prunes a given set of gradients. The file `corrupt_sgd.py` is for Question 2, and `autograd_hacks.py` is a helper file.

Question 1: Robust aggregator (5pts)

Let's simulate what a robust aggregator would do. Suppose you are given the following matrix X consisting of $n = 1545$ gradient vectors of dimension $d = 1536$ (load from `all_gradients.npy`). Suppose you know that the benign variance is $\|\Sigma\|_2 = 39275$, and decide to use a threshold of $\mathcal{E} = k\|\Sigma\|_2$ where $k = 9$. Please place any code you use to help answer the following questions in the `robust_aggregator.py` file.



```
>>> gradients
tensor([[-343.2032, -136.3944, -502.9129,  ...,  508.0136,  157.5610,  -67.0113],
        [-343.2016, -136.3937, -502.9129,  ...,  508.0101,  157.5605,  -67.0127],
        [-343.1989, -136.3892, -502.8905,  ...,  507.9893,  157.5609,  -67.0188],
        ...,
        [-342.0622, -135.4912, -500.7787,  ...,  505.9562,  156.6829,  -67.2080],
        [-342.7455, -135.6461, -501.8442,  ...,  506.9454,  157.0025,  -67.2080],
        [-343.2073, -136.3949, -502.9179,  ...,  508.0185,  157.5615,  -67.0114]])
```

1. Compute the covariance matrix Σ. What is the maximum variance? What is the direction of maximum variance (giving the first 3 and last 3 values is sufficient)? (2pt)

2. Compute the absolute distance of every gradient vector to the mean, with respect to the direction of max variance. Which gradient seems the most likely to be an outlier? Report the index number of this gradient (an integer in [0, 1544]), and give the first 3 and last 3 values of the gradient vector. (1pt)

3. Remove the gradient vector that seems most likely to be an outlier. Compute and report the maximum variance among the remaining gradient vectors. By how much did the maximum variance change? (1pt)

4. Repeat this procedure until max variance of the pruned set of gradients is below threshold of $\mathcal{E}$. How many poisoned gradients did you detect? Suppose I told you there were 673 poisoned gradients, what percentage did you detect? (1pt)

Evaluation Upload your answers to these questions in a PDF. Upload the `robust_aggregator.py` file containing all the code you wrote to help answer these questions.

Question 2: Adversarial corruption (5pts)

Now that we are familiar with the robust aggregation algorithm, switch gears and imagine being an adversary who wants to trick the robust aggregator. Consider the federated learning setting, where the global trainer gets gradients from several local users to update the global model. However, some of those gradients are maliciously corrupted by you.

We simulate this corruption of gradient vectors in the `corrupt_gradients` function in `corrupt_sgd.py`. For a batch of data, we compute the individual gradient vectors of each data sample in the batch, then corrupt an ϵ fraction of those gradients. Your task is to decide how you want to corrupt the gradients. The goal is to come up with a corruption mechanism that will

1. bypass the robust aggregation algorithm (i.e. your corrupted gradients won't get filtered out by the algorithm in Question 1), and
2. successfully diminish the model's training performance.

You can assume that the expected max variance of a benign set $||\Sigma|| = 39275$, and the robust aggregator uses a threshold of $\mathcal{E} = k||\Sigma||$ where $k = 9$.

For testing, you can train a model (`SST2_clean_model`) on a training dataset (`data/train.tsv`) using your implementation of the corrupted SGD and the robust aggregator (from Question 1). See if your gradient vectors gets filtered out. See if model performance is diminished: if the there was no poisoning, the training accuracy should reach 98% after 3 epochs, so if its train accuracy is still below 97% after 3 epochs of training, it is not ideal for the model trainer.

Evaluation Upload the completed `corrupt_sgd.py` file. For evaluation, your implementation of `corrupt_gradients` will be exported to run on `SST2_clean_model` and `data/train.tsv`. For fairness, the same implementation of robust aggregator from the TA will be used during evaluation. An epsilon value of 0.1 will be used. You get 1 pt if none of your gradient vectors gets filtered out, and 1 pt if the model's training accuracy falls below 97% after 3 epochs.

References

[1] Sarthak Choudhary, Aashish Kolluri, and Prateek Saxena. Attacking byzantine robust aggregation in high dimensions, 2024.