Phase 1

Mahdi Tabatabaei - 400101515
Heliya Shakeri - 400101391

Machine Learning

Dr. Amiri

July 3, 2024

Abstract

In the previous phase you familiarized yourself with some important topics in machine learning. In this phase you will do the implementations!

Machine Unlearning

In this question, you will be implementing SISA Algorithm and testing its performance.

Learning Phase

- **Sharding:** The training data is divided into S smaller subsets, or shards. Each shard contains a portion of the overall data. This ensures that the model training can be segmented into manageable parts.
- **Isolation:** Each shard is used to train a separate model or a component of a model independently. This isolation helps in minimizing the dependency across different parts of the training data.
- **Slicing:** Each shard is further divided into slices. Training occurs in a sequential manner, slice by slice, within each shard. This stepwise learning approach allows for finer control over the data's influence on the model. Specifically, each shard's data D_k is further uni formly partitioned into R disjoint slices such that $\bigcap_{i\in[R]} D_{k,i} = \emptyset$ and $\bigcup_{i\in[R]} D_{k,i} = D_k$: We perform training for e epochs to obtain M_k as follows:
  - At step 1, train the model using random initialization using only $D_{k,1}$, for e_1 epochs. Let us refer to the resulting model as $M_{k,1}$. Save the state of parameters associated with this model.
  - At step 2, train the model $M_{k,1}$ using $D_{k,1} \bigcup D_{k,2}, fore_2$ epochs. Let us refer to the resulting model as $M_{k,2}$. Save the parameter state.
  - At step R, train the model $M_{k,R1}$ using $\bigcup_{i\in[R]} D_{k,i} = D_k$ for e_R epochs. Let us refer to the resulting final model as $M_{k,R} = M_k$. Save the parameter state.
- **Aggregation:** The final model is constructed by aggregating the outputs from the independently trained shards. This aggregation helps in forming a comprehensive model while maintaining the benefits of isolation and segmentation.

Simulation Question 1. Implement the learning phase of the algorithm, and select the based models to be pretrained **ResNet 18** (you will need to add some fully connected layer(s) in order to make it suitable for the given dataset!). Use $S = 5, 10, 20$ and $R = 5, 10, 20$ and train different models on your data in each case, and report F1-score, accuracy, precision, recall and AUROC of all of your models with all of your proposed aggregation methods.

Method

At first, we implement the `load_data()` function to load, transform, and normalize our dataset, and to separate it into training and testing datasets. Next, we modify `ResNet-18` to fit the `CIFAR-10` dataset within the `create_model()` function. To implement the SISA algorithm, we utilize several key functions:

- The `train_shard` function handles the training process of a neural network model on a specific data shard over a set number of epochs, updating the model's parameters to minimize the loss.
- The `sisa_training` function divides the dataset into multiple shards and slices, trains a separate model on each slice, and returns a list of trained models. This approach enhances data privacy and model robustness by isolating the data and training models sequentially on smaller data subsets.

In this section, I primarily use average aggregation, although majority aggregation and other methods could also be employed. The aggregation part is handled in `evaluate_model`, which evaluates the F1-score, accuracy, precision, recall, and AUROC. To display these metrics, we use `display_metrics`.
During the training phase, I employ nine different conditions defined by combinations of $S = \{5, 10, 20\}$ and $R = \{5, 10, 20\}$. The metrics for each condition are then evaluated.

Result

I executed this code multiple times, and each run produced outputs that were similar, yet slightly different from each other. The output from one such run for training and test dataset is presented in the following tables:

Table 1: **Training** Results for Different Settings of S and R

S	R	**Accuracy (%)**	**Precision**	**Recall**	**F1 Score**	**AUROC**
5	5	86.50%	0.865	0.865	0.865	0.925
5	10	84.22%	0.843	0.842	0.842	0.912
5	20	81.02%	0.810	0.810	0.810	0.895
10	5	81.86%	0.819	0.819	0.818	0.899
10	10	78.33%	0.787	0.783	0.783	0.880
10	20	74.93%	0.753	0.749	0.750	0.861
20	5	75.92%	0.754	0.759	0.752	0.863
20	10	71.61%	0.718	0.716	0.715	0.842
20	20	68.73%	0.687	0.687	0.687	0.798

Table 2: **Testing** Results for Different Settings of S and R

S	R	**Accuracy (%)**	**Precision**	**Recall**	**F1 Score**	**AUROC**
5	5	84.12%	0.841	0.841	0.841	0.912
5	10	82.28%	0.822	0.823	0.822	0.901
5	20	79.39%	0.794	0.794	0.793	0.886
10	5	80.87%	0.809	0.809	0.808	0.894
10	10	77.68%	0.774	0.776	0.774	0.874
10	20	73.86%	0.736	0.739	0.736	0.853
20	5	74.54%	0.745	0.745	0.744	0.859
20	10	70.98%	0.709	0.710	0.707	0.838
20	20	65.34%	0.653	0.653	0.654	0.777

Unlearning Phase

When unlearning is required, the SISA algorithm allows for the selective removal of data by:

- **Targeting Specific Shards:** Since the training data is sharded, only the shards containing the data to be forgotten need to be retrained or adjusted. This significantly reduces the computational cost compared to retraining the entire model.
- **Updating Relevant Slices:** Within a targeted shard, k that is supposed to have some of its data removed, assume $M_{k,j}$ is the last model of the shard that all of its training data is not supposed to be removed. We then take a similar approach to the learning phase as we train $M_{k,j}$ using $\bigcup_{i<j+1} D'_{k,i}$ for e_R epochs, and so on like in the learning phase.($D'_{k,i}$'s are updated data slices with the specified data removed)
- **Efficient Re-Aggregation:** After updating the necessary shards and slices, the outputs are re-aggregated to form the updated model. This allows the model to effectively "forget" the specified data points while retaining the knowledge from the remaining data.

Simulation Question 2. Implement the unlearning algorithm on all your trained models in the previous question and "forget" 500 randomly chosen data of the whole dataset and measure the performance using your proposed metric, also report F1-score, accuracy, precision, recall and AUROC of all of your models with all of your proposed aggregation methods.

Method

First, we implement the `unlearn_data` function, which removes specific data points from a dataset by creating a new subset that excludes these points. This allows the model to "forget" the specified data. Then, we begin training the model with combinations of $S = \{5, 10\}$ and $R = \{5, 10\}$, resulting in four different conditions. The evaluation and other processes are conducted in the same manner as in previous parts.

Result

We show the result of evaluation and unlearning in following table.

Note: In all of my code executions, the metrics in this section did not exceed those of the previous section. However, here we observe higher metrics. Upon consulting with the teaching assistant, I was informed that this discrepancy might be due to the improper handling of the "forget data" process.

Table 3: **Testing** Results for Different Settings of S and R after unlearning

S	R	**Accuracy (%)**	**Precision**	**Recall**	**F1 Score**	**AUROC**
5	5	85.97%	0.863	0.859	0.860	0.922
5	10	84.01%	0.841	0.840	0.840	0.911
10	5	82.08%	0.822	0.820	0.820	0.900
10	10	78.37%	0.787	0.783	0.784	0.879

Evaluation

In this part we evaluate the effectiveness of the SISA unlearning algorithm, using Membership Inference Attack.

▬ *Membership Inference Attack*

A Membership Inference Attack aims to determine whether a particular data sample was part of the training dataset used to train a machine learning model. This type of attack leverages the model's responses to specific inputs, exploiting differences in the model's behavior between training data and unseen data. For instance, models often perform better on training data compared to unfamiliar data due to overfitting. By carefully analyzing the model's output probabilities, loss values, or other response characteristics, an attacker can infer the membership status of individual samples, potentially leading to privacy breaches and revealing sensitive information about the training data.

Simulation Question 3. Write a function that takes losses of two different datasets of your trained(but not unlearned) model,Computes cross-validation score of a Logistic Regressionbased Membership Inference Attack. Now, input the function with the losses of the 'forget set' and 500 randomly chosen data of the test set of the trained model and report the score. Do the same for the unlearned model. What did you expect the result to be for a perfectly unlearned model? How did the SISA Algorithm perform?

Method

Here, We implement `compute_losses` function. This function computes the loss values for each data point in a dataset using the specified model and returns these loss values as a NumPy array. This can be useful for analyzing model performance or for tasks such as membership inference attacks. After that we implement `mia_cross_val_score` function trains a logistic regression model to distinguish between training and test samples based on their loss values and calculates the cross-validation score for this model. This score indicates how well the logistic regression model can perform the membership inference attack.

For this section, you practically need to report four cross-validation scores; let's assume we take the first dataset as part of the training dataset that is not forgotten in the unlearn state, we take the second dataset as a part of the test, we take the third dataset as the training data that is forgotten in the unlearn state which is the same as forget, and we take the fourth dataset as another test set.

Now we need to report these:

- Cross-validation losses of datasets 1 and 2 for the model trained and not unlearned.
- Cross-validation losses of datasets 3 and 4 for the model trained and not unlearned.
- Cross-validation losses of datasets 1 and 2 for the model retrained and unlearned.
- Cross-validation losses of datasets 3 and 4 for the model retrained and unlearned.

Result

The MIA scores and the cross validation are reported in following table:

Table 4: MIA Scores for the Trained and Unlearned Models

Model	**Datasets 1 & 2**	**Datasets 3 & 4**
Trained Model	0.5302	0.5348
Unlearned Model	0.5276	0.5308

For a perfectly unlearned model, the expectation would be that the Membership Inference Attack (MIA) scores for datasets 3 and 4 should ideally be close to 0.5 or lower, indicating that the model does not retain specific information about the training data it was instructed to forget. This suggests that the model should be no better than random guessing at determining whether a given data point was in the training set, thus protecting privacy effectively.

From the MIA scores provided:

- **Datasets 1 and 2:** Both the trained and unlearned models have an MIA score of 0.99, indicating high vulnerability to membership inference attacks. This suggests that both models can easily determine whether data points were part of these datasets, which likely were not targeted by the unlearning process.
- **Datasets 3 and 4:** The trained model has an MIA score of 0.5348, while the unlearned model has a slightly lower score of 0.5308.

The scores for datasets 3 and 4, which include the "forgotten" data, show that the unlearned model (MIA score of 0.5308) performed slightly better in terms of reducing the risk of membership inference compared to the trained model (MIA score of 0.5348). However, the difference is minimal, suggesting that the unlearning process had a marginal effect.

The small decrease in the MIA score for the unlearned model suggests that while there was some effect from the unlearning process, it was not substantial. For a perfectly unlearned model, a significant reduction in MIA scores would have been expected, ideally approaching 0.5.

The SISA algorithm, in this context, seems to have performed below expectations for effective data unlearning. While there was some reduction in the MIA score, the change was not significant enough to suggest that the data was effectively "forgotten."

This analysis points to the need for further optimization or a reevaluation of the SISA algorithm's unlearning strategy to enhance its effectiveness in reducing the model's memory of the "forgotten" data, thereby improving privacy protection against membership inference attacks.

Add On, Evaluation

To evaluate the effectiveness of the SISA unlearning algorithm, we will perform a backdoor attack on the model and assess the model's performance before and after unlearning.

Backdoor Attack

A backdoor attack involves poisoning the training data so that the model learns to associate a specific trigger (e.g., a pattern or mark) with a target label. During inference, the presence of this trigger in any input data will cause the model to misclassify it as the target label.

Assessing the Attack

- Evaluate the model on clean test data to ensure general performance is not significantly degraded.
- Evaluate the model on test data containing the backdoor trigger to measure the success rate of the backdoor attack.

▬ *Measuring the Success of the Backdoor Attack*

The success of the backdoor attack can be measured by the attack success rate (ASR), which is the percentage of test samples containing the backdoor trigger that are incorrectly classified as the target class. High ASR indicates a successful attack.

Add On. Simulation Question 1. Randomly select 500 data points from a specific class in the training set. For each selected data point, turn a 3x3 block of pixels to black, with the position of the block chosen randomly. Train your best model of the previous questions, using the poisoned dataset. Evaluate the model's performance, using all the same metrics as before, on clean test data to ensure general performance is not significantly degraded. Also, calculate the attack success rate (ASR) as the percentage of test samples misclassified as the target class.

Add On. Simulation Question 2. Unlearn the same 500 data, and evaluate the model's performance, using all the same metrics ass before, on clean test data to ensure general performance is not significantly degraded. Also, calculate the attack success rate (ASR) as the percentage of test samples misclassified as the target class.

Method

We use following functions for this section:

- `select_and_poison_data_inplace`: Selects 500 data points from a specified class within the training set and applies a "poisoning" process by blocking out a part of the image data, effectively modifying the dataset in-place to include these alterations.
- `predict_models`: Evaluates a list of models on a given dataset, typically a test set, by predicting outcomes and aggregating the results across the entire dataset for further analysis.
- `aggregate_average`: Aggregates predictions from multiple models by averaging their predicted probabilities, followed by determining the final predicted class based on the highest average probability.
- `poison_dataset`: Generates a new dataset where images from non-target classes are poisoned, aiming to mislead a trained model when this dataset is used.
- `calculate_asr (Attack Success Rate)`: Computes the effectiveness of the poisoning by calculating the proportion of poisoned images that are misclassified as the target class by the model.
- `calculate_poisoned_asr`: Similar to *calculate_asr*, but specifically applies to a scenario where the dataset used includes poisoned images from various classes, assessing the impact of widespread data poisoning on model accuracy.

We use these function for our bes model which was the condition that $S = 5, R = 5$ satisfies.

Results

To represent the Attack Success Rate (ASR) values before and after poisoning, the table has come.

Table 5: Comparison of Attack Success Rates (ASR) Before and After Poisoning

Condition	Before Poisoning	After Poisoning	Change (%)
Normal Model	12.58%	64.44%	+51.86%
Model with Unlearning	12.11%	3.33%	-8.78%

Two Following tables and comparing them with previous parts indicates that general performance is not significantly degraded.

Table 6: Performance Metrics on the Testing Dataset

Dataset	Accuracy	Precision	Recall	F1 Score	AUROC
Testing	0.8597	0.863	0.8597	0.86	0.922

Table 7: Performance Metrics on the Testing Dataset

Dataset	Accuracy	Precision	Recall	F1 Score	AUROC
Testing	0.838	0.8388	0.838	0.8373	0.91

Private Training

In this section, you will first train a classification model using a standard approach, then train it with privacy enhancements, and compare the MIA accuracy of both models. (Use the provided model in model.py for all tasks.)

Simulation Question 4. Use 80 percent of the CIFAR-10 training data to train your model. This will serve as your baseline model.

Method

Dataset Preparation: For the task of training a baseline model on CIFAR-10, the dataset was first imported using the `torchvision.datasets` module. The dataset consists of 50,000 training images and 10,000 test images across 10 classes. To prepare the data for training and validation, we applied a series of transformations: converting images to tensors and normalizing them with pre-defined mean and standard deviation values for each channel. We partitioned the training data into two subsets: 80% (40,000 images) was used for training the model, and the remaining 20% (10,000 images) served as a validation set. This split was accomplished using the `random_split` function provided by PyTorch.

Model Setup: The model used in this simulation, referred to as `CIFAR10Classifier`, was defined in a separate Python file (`model.py`). It was loaded into the training script where it was initialized and moved to a CUDA-capable GPU for accelerated computation. The choice of device was conditional, depending on the availability of GPU resources, with a fallback to CPU if necessary.

Training Process: Training involved setting the model to training mode and iterating over the training dataset using mini-batches. For each batch, we performed the following steps:

1. Forward pass: Compute the model's prediction output.
2. Compute loss: Calculate the cross-entropy loss between the predicted outputs and the actual labels.
3. Backward pass: Perform backpropagation to compute gradients.
4. Update model parameters: Adjust the model weights using the Adam optimizer with a learning rate of 0.001.

This process was repeated for 10 epochs, with intermittent logging to monitor the training loss after every 200 mini-batches.

Validation: Post-training, the model was evaluated on the validation set to assess its performance. This evaluation was done in the model's evaluation mode, which disables features like dropout to ensure consistency in inference. The validation phase involved computing both the loss and the accuracy, providing a clear measure of model performance without training interference.

Results

Performance Metrics During Training: The following table presents the training loss and validation accuracy for each epoch, illustrating the model's learning progression over the 10 epochs:

Table 8: Validation Loss and Accuracy In each Epoch

Epoch	Validation Loss	Validation Accuracy (%)
1	1.4201	50.89
2	1.2809	55.56
3	1.1824	59.26
4	1.1526	60.41
5	1.1153	62.10
6	1.0989	62.67
7	1.0559	63.55
8	1.0449	63.93
9	1.0375	64.14
10	1.0371	64.26

Performance Metrics on the Testing Dataset: After completing the training, the model was evaluated on the test dataset to assess its generalization capability. The

Accuracy was **63.84** percent which seems reasonable.

Simulation Question 5. Train your baseline model with privacy enhancements. This is your modified model. Ensure that the test accuracy difference between your baseline model and the modified model is less than 15

Method

Model Initialization: The `CIFAR10Classifier` model was initialized and trained using three privacy-enhancement techniques: adversarial training, differential privacy, and the PATE (Private Aggregation of Teacher Ensembles) framework.

Adversarial Training: To enhance the model's robustness against adversarial attacks, the following steps were implemented:

1. Adversarial Example Generation: For each training batch, adversarial examples were generated using the FGSM (Fast Gradient Sign Method) attack with a perturbation amount (`epsilon`) of 0.1.

2. Training Process: The model was trained on both the original and adversarial examples. For each batch, the loss was calculated separately for the original and adversarial inputs, and their average was used for backpropagation and parameter updates.

3. Validation: After each epoch, the model's performance was evaluated on the validation set, and metrics such as validation loss and accuracy were recorded.

Differential Privacy: To ensure differential privacy during training, the following procedures were adopted:

1. Privacy Engine Initialization: A privacy engine was attached to the optimizer, which introduced noise into the gradients and enforced a maximum gradient norm.

2. Noise Addition: Gaussian noise was added to the inputs during training to obscure individual data points.

3. Temperature Scaling: Outputs were scaled using a temperature parameter to stabilize training.

4. Training and Validation: The model was trained for 10 epochs with the modified optimizer and privacy enhancements, and performance metrics were recorded after each epoch.

PATE Framework: The PATE framework was used to train a student model based on the aggregated outputs of multiple teacher models:

1. Teacher Model Training: Ten teacher models were trained independently on disjoint subsets of the CIFAR-10 training data.

2. Aggregation of Teacher Predictions: Predictions from the teacher models were aggregated and subjected to noise addition to generate the labels for training the student model.

3. Student Model Training: The student model was trained on the dataset created from the aggregated noisy predictions of the teacher models. The training process included standard optimization and performance monitoring.

Evaluation: The performance of each privacy-enhanced model was evaluated on the test dataset. Accuracy metrics were calculated and compared to ensure that the difference in test accuracy between the baseline and the modified models was within the acceptable range of less than 15%.

Results

Adversarial Training: The model was trained with adversarial examples for 10 epochs. The final test accuracy of the adversarially trained model was **64.55%**. The following table presents the validation loss and validation accuracy for each epoch:

Table 9: Adversarial Training - Validation Performance Metrics

Epoch	Validation Loss	Validation Accuracy (%)
1	1.4330	49.18
2	1.2781	55.71
3	1.1776	58.40
4	1.1632	59.87
5	1.1174	61.21
6	1.0866	61.87
7	1.0852	62.24
8	1.0569	63.71
9	1.0375	64.14
10	1.0337	64.16

Differential Privacy:s The model was trained with differential privacy enhancements for 10 epochs. The final test accuracy of the differentially private model was **38.41%**. Table 10 presents the validation loss and validation accuracy for each epoch:

PATE Framework: The PATE framework was used to train the student model, which achieved a test accuracy of **49.66%**.
Among the three privacy-enhancement methods tested, the PATE framework demonstrated the most promising results with a test accuracy of 49.66%. This accuracy gap compared to the baseline model (which had an accuracy of 64.55%) is less than 15%, making PATE a viable method for achieving privacy without significantly degrading the model's performance.

.

Table 10: Differential Privacy Training - Validation Performance Metrics

Epoch	Validation Loss	Validation Accuracy (%)
1	2.0943	26.24
2	1.9811	29.84
3	1.9138	32.48
4	1.8793	33.66
5	1.8392	34.86
6	1.8117	35.63
7	1.8010	36.00
8	1.7758	37.00
9	1.7524	37.38
10	1.7539	37.76

Table 11: Test Accuracy of Models with Privacy Enhancements

Method	Test Accuracy (%)
Adversarial Training	64.55
Differential Privacy	38.41
PATE	49.66

Simulation Question 6. Train two Attacker Models based on MIA techniques learned in Phase 0, one for the baseline model and one for the modified model. Compare the MIA accuracy of these two attacker models. Use 80 percent of the training data as your seen data, and the remaining training data along with the test data as your unseen data.

Method

Data Preparation: We split the CIFAR-10 training dataset into seen and unseen datasets:

- Seen Dataset: 80% of the training data.
- Unseen Dataset: The remaining 20% of the training data, combined with the CIFAR-10 test data to form the final unseen dataset.

We created data loaders for both the seen and unseen datasets to facilitate batch processing during training and evaluation.

Training Shadow Models: To create a realistic MIA scenario, we trained ten shadow models independently on disjoint subsets of the CIFAR-10 training data:

- We used the same architecture as the baseline model (`CIFAR10Classifier`) for each shadow model.
- We trained each shadow model using the Adam optimizer with a learning rate of 0.001 and a weight decay of 1e-4.

- Each shadow model was trained for 10 epochs.

Generating Attacker Datasets: We generated datasets for training the attacker models using the shadow models:

- Seen Data Predictions: We obtained predictions from the shadow models on the seen dataset.
- Unseen Data Predictions: We obtained predictions from the shadow models on the unseen dataset.
- We created the attacker datasets by concatenating the predictions from the seen and unseen datasets, with corresponding labels (1 for seen data, 0 for unseen data).

Training Attacker Models: We trained attacker models to distinguish between seen and unseen data:

- Model Architecture: The attacker model consisted of two fully connected layers with ReLU activation in the first layer and a sigmoid activation in the output layer.
- Training Process: We trained the attacker models using the generated attacker datasets, with binary cross-entropy loss and the Adam optimizer.
- Each attacker model was trained for 10 epochs.

Evaluating Attacker Models: We evaluated the performance of the attacker models on the attacker datasets:

- Accuracy Calculation: We calculated the accuracy of the attacker models as the percentage of correctly classified samples (seen vs. unseen).
- The evaluation was conducted separately for the attacker models trained on the baseline model and the modified model.

Results

Attacker Model Training: The attacker models for both the baseline and modified models were trained for 10 epochs. The training losses for each epoch are presented in Table 12.

Attacker Model Evaluation: The performance of the attacker models was evaluated on the attacker datasets. The accuracy of the attacker models in distinguishing between seen and unseen data is presented in table 13.

Analysis: The accuracy of the attacker models for both the baseline and modified models was **66.67%**. In the context of Membership Inference Attacks (MIA), a lower accuracy for the attacker model is desirable as it indicates better privacy preservation. However, in this case, the attacker model achieved the same accuracy for both the baseline and modified models. This result suggests that there might be an issue with our implementation or the training process of the attacker models, as we would expect the

Table 12: Attacker Model Training Loss

Epoch	Baseline Model Loss	Modified Model Loss
1	0.6399	0.6397
2	0.6380	0.6375
3	0.6373	0.6370
4	0.6369	0.6367
5	0.6366	0.6365
6	0.6364	0.6364
7	0.6363	0.6363
8	0.6363	0.6362
9	0.6362	0.6362
10	0.6361	0.6361

Table 13: Attacker Model Evaluation Accuracy

Model	Attacker Model Accuracy (%)
Baseline Model	66.67
Modified Model	66.67

accuracy to be lower for the modified model if it had better privacy-preserving properties. having tried many ways, the result was consistent. So, further investigation is needed to identify and resolve the discrepancy to ensure the modified model provides enhanced privacy. Based on the article 'Membership Inference Attacks against Machine Learning Models', we tried another approach as below:

Shadow Model Training: To simulate membership inference attacks as described by Shokri et al., we trained multiple shadow models. These shadow models mimic the behavior of the target model under similar conditions:

- Shadow Model Definition: We used a convolutional neural network architecture for the shadow models, similar to the target model, to ensure realistic simulation.
- Training Shadow Models: We divided the training data equally among several shadow models. We trained each shadow model independently using its respective subset of the training data, capturing the diversity in training data distribution.
- Data Collection for Attack Models: After training, we collected the outputs (softmax probabilities) of the shadow models for both seen (training) and unseen (validation and test) data. We labeled this data accordingly (1 for seen data, 0 for unseen data), creating the datasets needed to train the attack models.

Attack Model Training: We trained attack models to perform membership inference based on the outputs of the shadow models, following the approach proposed by Shokri et al.:

- Attack Model Definition: We used a neural network with fully connected layers as the attack model. This model was trained to differentiate between outputs from the shadow models corresponding to seen and unseen data.

- Training Attack Models: We trained separate attack models for each class using the collected shadow model data. The training data for each attack model consisted of the outputs from the shadow models along with the corresponding membership labels. By training the attack models on these labeled outputs, they learned to infer membership status from the output distributions.

Evaluation: We evaluated the effectiveness of the attack models by comparing their performance on the baseline and private models as follows:

- **Membership Inference Attack:** We used the trained attack models to predict the membership status of the data points in the baseline and private models. We measured the attack models' performance by calculating the accuracy of these predictions, which indicates how well the models can distinguish between seen and unseen data points.
- **Comparison:** We compared the membership inference accuracy between the baseline model and the private model to assess the impact of the privacy-preserving techniques. A lower accuracy on the private model would indicate that the privacy-preserving techniques are effective in mitigating membership inference attacks.

Evaluation of Attack Models: We evaluated the effectiveness of the attack models by comparing their performance on the baseline and private models. The attack models' performance is measured by the accuracy of their membership inference predictions.

Table 14: Membership Inference Attack (MIA) Accuracy

Model	Attacker Model Accuracy (%)
Baseline Model	63.63
Modified Model	53.45

- **Baseline Model:** The attack model achieved a membership inference accuracy of 64.00%. This indicates that the attack model was able to correctly identify whether data points were part of the training set 64% of the time.
- **Private Model:** The attack model achieved a membership inference accuracy of 53.00%. The lower accuracy compared to the baseline model suggests that the privacy-preserving techniques employed in the private model were effective in mitigating membership inference attacks.

Simulation Question 7. Improve your attacker models to achieve better MIA accuracy for both the baseline and modified models (e.g., by increasing the number of shadow models). Then compare the new accuracies with the previous results.

Method

In this question, we aim to improve the performance of our attacker models to achieve better Membership Inference Attack (MIA) accuracy for both the baseline and modified models. Several enhancements are considered to potentially boost the MIA accuracy:

1. **Increase the Number of Shadow Models**: By increasing the number of shadow models, we can generate a richer and more diverse dataset for training the attacker models. This diversity can help the attacker models learn better representations and improve their accuracy in distinguishing between seen and unseen data.
2. **Use More Sophisticated Attacker Models**: Instead of using a simple neural network architecture, we can explore more complex architectures for the attacker models. Advanced architectures such as convolutional neural networks (CNNs) or deeper fully connected networks might capture more nuanced patterns in the data.
3. **Data Augmentation**: Applying data augmentation techniques to the datasets used to train the shadow models can enhance the generalizability of the attacker models. Techniques such as random cropping, flipping, and rotation can be employed to artificially expand the training dataset.
4. **Hyperparameter Tuning**: Experimenting with different hyperparameters for both the shadow models and the attacker models can lead to improved performance. Hyperparameters such as learning rate, batch size, and the number of epochs can be fine-tuned to optimize the training process.

We implemented these enhancements and compared the new accuracies with the previous results to evaluate the effectiveness of the improvements.

Membership Inference Attack

Simulation Question 8. Attempt to train an attacker model for the given private model (`private_model.pth`). We will test it on our dataset during the online presentation session. A competitive bonus point is available for the best performance.

Method

To implement this section, we utilized code similar to what was used in the previous part, but applied differently. Initially, we attempted to use code provided by the teaching assistant and shared on the communication channel. However, I encountered difficulties in using this code effectively, prompting me to develop my own implementation. My code successfully replicated the same performance metrics as the teaching assistant's version, ensuring consistency in our results.

We employed shadow model training to simulate attack scenarios. Due to constraints in GPU resources and time, we were only able to train five shadow models. It is likely that training more shadow models could significantly improve the performance of our attacker model by providing a richer approximation of the target model's decision boundaries.

Regarding the metrics for the private model, there are concerns about the accuracy of the results. The confusion matrix generated from my implementation does not seem correct, which suggests potential issues with the model's performance or possibly with the data handling. This discrepancy raises questions about the effectiveness of the shadow training under limited resource conditions and the fidelity of the model evaluations.

This situation underscores the importance of thorough validation and testing of machine learning models, especially in privacy-sensitive applications. It also highlights the

need for adequate computational resources to achieve more reliable and robust model assessments.

Result

Table 15: Performance Metrics and Confusion Matrix

Metric	**Value**	
	Predicted No	Predicted Yes
Confusion Matrix	0	0
	3498	6502
Precision	1.0000	
Recall	0.6502	
F1 Score	0.7880	
Accuracy	0.6502	

Scripts

Simulation Question 1.

```python
def load_data():
    transform = transforms.Compose([
        transforms.Resize((224, 224)),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
    ])
    train_set = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
    test_set = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform)
    return train_set, test_set
```

Source Code 1: Data Prepration

```python
# Modify ResNet-18 for CIFAR-10 (10 classes)
def create_model():
    model = resnet18(pretrained=True)
    num_features = model.fc.in_features
    model.fc = torch.nn.Linear(num_features, 10)  # CIFAR-10 has 10 classes
    model = model.to(device)  # Move model to GPU
    return model
```

Source Code 2: Model Prepration

```python
# SISA Algorithm
def train_shard(model, data_loader, epochs):
    criterion = torch.nn.CrossEntropyLoss().to(device)
    optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
    model.train()
    for epoch in range(epochs):
        for inputs, labels in data_loader:
            inputs, labels = inputs.to(device), labels.to(device)  # Move data to GPU
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
    return model

def sisa_training(dataset, S, R, epochs_list):
    shard_size = len(dataset) // S
    shards = random_split(dataset, [shard_size] * S)  # Creating shards
    models = []

    for shard in shards:
        slice_size = len(shard) // R
        slices = random_split(shard, [slice_size] * R)  # Creating slices
        model = create_model()
        for i, slice in enumerate(slices):
            data_loader = DataLoader(slice, batch_size=32, shuffle=True)
            model = train_shard(model, data_loader, epochs_list[i % len(epochs_list)])
```

```
        models.append(model)
    return models
```

Source Code 3: SISA Algorithm

```
# Evaluation and Aggregation
def evaluate_model(models, train_loader, test_loader):
    def metrics_report(loader):
        total_preds = []
        total_labels = []
        with torch.no_grad():
            for data, target in loader:
                data, target = data.to(device), target.to(device)
                model_outputs = [model(data).detach() for model in models]
                avg_output = torch.mean(torch.stack(model_outputs), dim=0)
                _, predicted = torch.max(avg_output, 1)
                total_preds.extend(predicted.cpu().tolist())
                total_labels.extend(target.cpu().tolist())

        accuracy = accuracy_score(total_labels, total_preds)
        precision = precision_score(total_labels, total_preds, average='macro')
        recall = recall_score(total_labels, total_preds, average='macro')
        f1 = f1_score(total_labels, total_preds, average='macro')
        true_binary = label_binarize(total_labels, classes=np.unique(total_labels))
        pred_binary = label_binarize(total_preds, classes=np.unique(total_labels))
        auroc = roc_auc_score(true_binary, pred_binary, multi_class='ovr')

        return accuracy, precision, recall, f1, auroc

    # Optionally evaluate on training data
    results = {}

    test_metrics = metrics_report(test_loader)
    results["Testing"] = test_metrics

    return results
```

Source Code 4: Aggregation and Evaluation

```
# Function to display metrics in a beautiful table
def display_metrics(results):
    headers = ["Dataset", "Accuracy", "Precision", "Recall", "F1 Score", "AUROC"]
    table = []
    for dataset, metrics in results.items():
        row = [dataset] + [f"{metric:.4f}" for metric in metrics]
        table.append(row)
    print(tabulate(table, headers=headers, tablefmt="grid"))
```

Source Code 5: Displaying F1-score, accuracy, precision, recall and AUROC

```
# Number of Shards and Slices
```

```
S_values = [5, 10, 20]  # List of S values
R_values = [5, 10, 20]  # List of R values
epochs = [2, 2, 5]  # Number of epochs for training

best_accuracy = 0        # Variable to track the best accuracy
best_model = None        # Variable to store the best model
best_train_loader = None # Variable to store the best train loader
best_test_loader = None  # Variable to store the best test loader
best_result = {}         # Variable to store the best result

for S in S_values:
    for R in R_values:
        print(f"\nTraining with S={S}, R={R}")

        # Train the model with the given S, R, and epochs
        model = sisa_training(train_set, S, R, epochs)

        # Evaluate the trained model
        train_loader = DataLoader(train_set, batch_size=32, pin_memory=True, num_workers=4)
        test_loader = DataLoader(test_set, batch_size=32, pin_memory=True, num_workers=4)
        result = evaluate_model(model, train_loader, test_loader)

        # Display evaluation metrics
        display_metrics(result)

        # Calculate the accuracy score
        accuracy = result["Testing"][0]

        # Update the best model if the current one is better
        if accuracy > best_accuracy:
            best_accuracy = accuracy
            best_model = model
            best_train_loader = train_loader
            best_test_loader = test_loader
            best_result = result
```

Source Code 6: Training Phase of Simulation Question 1.

Simulation Question 2.

```
# Function to unlearn the data
def unlearn_data(dataset, forget_indices):
    # Create a subset of the dataset excluding the forget_indices
    retain_indices = list(set(range(len(dataset))) - set(forget_indices))
    return Subset(dataset, retain_indices)
```

Source Code 7: Unlearning Data

```
# Number of Shards and Slices
S_values = [5, 10]
R_values = [5, 10]
epochs = [2, 2, 5]  # Simplified for demonstration

best_accuracy_after_unlearning = 0        # Variable to track the best accuracy
best_model_after_unlearning = None        # Variable to store the best model
```

```
best_train_loader_after_unlearning = None # Variable to store the best train loader
best_test_loader_after_unlearning = None  # Variable to store the best test loader
best_result_after_unlearning = {}         # Variable to store the best result

# Select 500 random samples to forget
forget_indices = random.sample(range(len(train_set)), 500)

# Unlearn 500 samples
unlearned_train_set = unlearn_data(train_set, forget_indices)

for S in S_values:
    for R in R_values:
        print(f"\nTraining with S={S}, R={R}")

        # Train the model with the given S, R, and epochs
        model_after_unlearning = sisa_training(unlearned_train_set, S, R, epochs)

        # Evaluate the trained model
        train_loader_after_unlearning = DataLoader(unlearned_train_set, batch_size=32, pin_memory=True, num_workers=4)
        test_loader_after_unlearning = DataLoader(unlearned_train_set, batch_size=32, pin_memory=True, num_workers=4)
        result_after_unlearning = evaluate_model(model_after_unlearning, train_loader_after_unlearning, test_loader_after_unlearning)

        # Display evaluation metrics
        display_metrics(result_after_unlearning)

        # Calculate the accuracy score
        accuracy_after_unlearning = result_after_unlearning["Testing"][0]

        # Update the best model if the current one is better
        if accuracy_after_unlearning > best_accuracy_after_unlearning:
            best_accuracy_after_unlearning = accuracy_after_unlearning
            best_mode_after_unlearningl = model_after_unlearning
            best_train_loader_after_unlearning = train_loader_after_unlearning
            best_test_loader_after_unlearning = test_loader_after_unlearning
            best_result_after_unlearning = result_after_unlearning
```

Source Code 8: Training Phase of Simulation Question 2.

Simulation Question 3.

```
# Function to compute losses
def compute_losses(model, data_loader):
    model.to(device)
    model.eval()
    losses = []
    with torch.no_grad():
        for inputs, labels in data_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
```

```
            loss = cross_entropy(outputs, labels, reduction='none')
            losses.extend(loss.cpu().numpy())
    return np.array(losses)
```

Source Code 9: Computing Loss

```
# Function to compute cross-validation score of MIA
def mia_cross_val_score(train_losses, test_losses):
    labels = np.concatenate((np.ones(len(train_losses)), np.zeros(len(test_losses))))
    losses = np.concatenate((train_losses, test_losses)).reshape(-1, 1)
    clf = LogisticRegressionCV(cv=5, random_state=42, max_iter=1000).fit(losses, labels)
    scores = cross_val_score(clf, losses, labels, cv=5)
    return scores.mean()
```

Source Code 10: Compute MIA

```
# Select 500 random samples to forget
retain_indices = list(set(range(len(train_set))) - set(forget_indices))

retain_set = Subset(train_set, retain_indices)
forget_set = Subset(train_set, forget_indices)
test_sample_indices_1 = random.sample(range(len(test_set)), 500)
test_sample_set_1 = Subset(test_set, test_sample_indices_1)
test_sample_indices_2 = random.sample(range(len(test_set)), 500)
test_sample_set_2 = Subset(test_set, test_sample_indices_2)

retain_loader = DataLoader(retain_set, batch_size=32, shuffle=False)
forget_loader = DataLoader(forget_set, batch_size=32, shuffle=False)
test_sample_loader_1 = DataLoader(test_sample_set_1, batch_size=32, shuffle=False)
test_sample_loader_2 = DataLoader(test_sample_set_2, batch_size=32, shuffle=False)

# Assuming models and models_after_unlearning are loaded or defined elsewhere
# Compute losses for the trained model
retain_losses = np.concatenate([compute_losses(model,  DataLoader(retain_set, batch_size=32, shuffle=False)) for model in best_model])
forget_losses = np.concatenate([compute_losses(model, DataLoader(forget_set, batch_size=32, shuffle=False)) for model in best_model])
test_losses_1 = np.concatenate([compute_losses(model, DataLoader(test_sample_set_1, batch_size=32, shuffle=False)) for model in best_model])
test_losses_2 = np.concatenate([compute_losses(model, DataLoader(test_sample_set_2, batch_size=32, shuffle=False)) for model in best_model])

mia_cross_val_score12 = mia_cross_val_score(retain_losses, test_losses_1)
mia_cross_val_score34 = mia_cross_val_score(forget_losses, test_losses_2)

# MIA scores for the trained model
print("MIA scores for the trained model:")
print("Datasets 1 and 2:", mia_cross_val_score12)
print("Datasets 3 and 4:", mia_cross_val_score34)

# Compute losses for the unlearned model
```

```
retain_losses_after = np.concatenate([compute_losses(model, DataLoader(
    retain_set, batch_size=32, shuffle=False)) for model in
    models_after_unlearning])
forget_losses_after = np.concatenate([compute_losses(model, DataLoader(
    forget_set, batch_size=32, shuffle=False)) for model in
    models_after_unlearning])
test_losses_1_after = np.concatenate([compute_losses(model, DataLoader(
    test_sample_set_1, batch_size=32, shuffle=False)) for model in
    models_after_unlearning])
test_losses_2_after = np.concatenate([compute_losses(model, DataLoader(
    test_sample_set_2, batch_size=32, shuffle=False)) for model in
    models_after_unlearning])

mia_cross_val_score_after12 = mia_cross_val_score(retain_losses_after,
    test_losses_1_after)
mia_cross_val_score_after34 = mia_cross_val_score(forget_losses_after,
    test_losses_2_after)

# MIA scores for the unlearned model
print("MIA scores for the unlearned model:")
print("Datasets 1 and 2:", mia_cross_val_score_after12)
print("Datasets 3 and 4:", mia_cross_val_score_after34)
```

Source Code 11: Cross Validation and MIA for Simulation Question 3.

Add On. Simulation Question 1.

```
#  Select 500 data points from a specific class in the training set
def select_and_poison_data_inplace(dataset, target_class, num_samples
    =500, block_size=3):
    class_indices = [i for i in range(len(dataset)) if dataset.targets[i]
     == target_class]
    selected_indices = random.sample(class_indices, num_samples)
    block_size = int(224 / 32) * block_size
    if block_size != 0:
        for idx in selected_indices:
            #print(dataset.targets[idx])
            img = dataset.data[idx]
            img = np.copy(img)  # To avoid modifying the original image
            # Randomly select a position for the 3x3 block
            x = random.randint(0, img.shape[0] - block_size)
            y = random.randint(0, img.shape[1] - block_size)
            img[x:x+block_size, y:y+block_size, :] = 0  # Turn the block
    to black
            dataset.data[idx] = img

    return selected_indices
```

Source Code 12: Selecting Poisoned Indices

```
def predict_models(models, test_loader):
    all_predictions = []
    for model in models:
        model.eval()  # Set model to evaluation mode
        model_predictions = []
        for inputs, _ in test_loader:
            inputs = inputs.to(device)
            with torch.no_grad():
```

```
                pred = model(inputs)
            model_predictions.append(pred.cpu())
        all_predictions.append(torch.cat(model_predictions))
    return all_predictions
```

Source Code 13: Predicting Mpdels' Probability

```
def aggregate_average(models, test_loader):
    all_predictions = predict_models(models, test_loader)
    averaged_probabilities = torch.mean(torch.stack(all_predictions), dim=0)
    aggregated_predictions = torch.argmax(averaged_probabilities, dim=1)
    averaged_probabilities = F.softmax(averaged_probabilities, dim=1)
    return aggregated_predictions, averaged_probabilities

    def poison_dataset(dataset, num_samples, target_class=0):
    # Prepare poisoned images from other classes
    poisoned_datasets = []

    for i in range(1, 10):
        if i != target_class:
            poisoned_indices = select_and_poison_data_inplace(dataset, target_class=i, num_samples=num_samples, block_size=3)
            poisoned_subset = Subset(dataset, poisoned_indices)
            poisoned_datasets.append(poisoned_subset)

    # Concatenate all poisoned subsets into one dataset
    poisoned_dataset = ConcatDataset(poisoned_datasets)

    return poisoned_dataset
```

Source Code 14: Aggregation to Mean

```
# Calculate the Attack Success Rate (ASR)
def calculate_asr(dataset, model, num_samples, target_class=0):
    data_loader = DataLoader(dataset, batch_size=32, shuffle=False)

    aggregated_predictions, _ = aggregate_average(model, data_loader)

    misclassified_count = (aggregated_predictions == target_class).sum().item()
    total_count = len(dataset)

    asr = misclassified_count / total_count

    return asr
```

Source Code 15: Calculate ASR for Non-Poisoned Dataset

```
    # Poisening the Best model; S = 5, R = 5
poisoned_model = best_model

# Evaluate the poisoned model's performance on clean test data
display_metrics(best_result)

# Calculate the attack success rate (ASR)
asr = calculate_asr(test_set, poisoned_model, 100, 0)
print(f"Attack Success Rate (ASR) Before Poisoninng: {asr * 100:.2f}%")
```

```
10
11 # Calculate the attack success rate (ASR)
12 asr = calculate_poisoned_asr(test_set, poisoned_model, 100, 0)
13 print(f"Attack Success Rate (ASR) After Poisoninng: {asr * 100:.2f}%")
```

Source Code 16: ASR for Non-Poisoned Dataset for Add On. Simulation Question 1.

Add On. Simulation Question 2.

```
1 # Calculate the Attack Success Rate (ASR)
2 def calculate_poisoned_asr(dataset, model, num_samples, target_class=0):
3     # Prepare poisoned images from other classes
4     poisoned_dataset = poison_dataset(dataset, num_samples, target_class
      =0)
5     poisoned_loader = DataLoader(poisoned_dataset, batch_size=32, shuffle
      =False)
6
7     aggregated_predictions, _ = aggregate_average(model, poisoned_loader)
8
9     misclassified_count = (aggregated_predictions == target_class).sum().
      item()
10    total_count = len(poisoned_dataset)
11
12    asr = misclassified_count / total_count
13
14    return asr
```

Source Code 17: Calculate ASR for Poisoned Dataset

```
1 # Poisening the Best model; S = 5, R = 5
2 poisoned_model_after_unlearning = best_mode_after_unlearningl
3
4 # Evaluate the poisoned model's performance on clean test data
5 display_metrics(best_result_after_unlearning)
6
7 # Calculate the attack success rate (ASR)
8 asr = calculate_asr(test_set, poisoned_model_after_unlearning, 100, 0)
9 print(f"Attack Success Rate (ASR) with unlearning Before Poisoninng: {asr
      * 100:.2f}%")
10
11 # Calculate the attack success rate (ASR)
12 asr = calculate_poisoned_asr(test_set, poisoned_model_after_unlearning,
      100, 0)
13 print(f"Attack Success Rate (ASR) with unlearning After Poisoninng: {asr
      * 100:.2f}%")
```

Source Code 18: ASR for Poisoned Dataset for Add On. Simulation Question 2.