Report: Homework 4

Mao Chuan, 2300013218

June 17, 2025

1 MIA Attack

In this part, I implemented the Membership Inference Attack (MIA), and the final accuracy achieved was 89.19%, as shown in Figure 1.

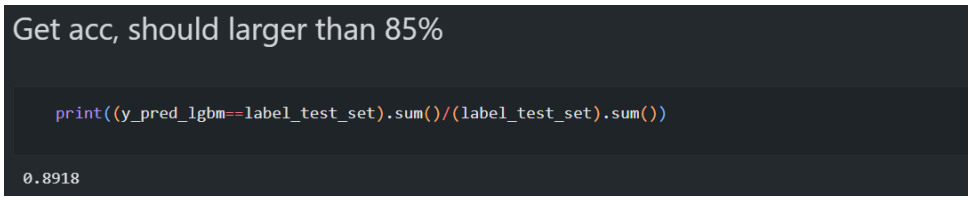


Figure 1: Final accuracy of MIA attack

1.1 Create Dataset

(2 points) To implement the MIA attack, I created a dataset (size N) in the form $\{x_i, y_i\}_{i=1}^N = \{logits_i, bool_i\}_{i=1}^N$, where $logits_i$ is the model's output for input image i, and $bool_i$ is a binary variable indicating whether image i was used to train the model (1 if yes, 0 otherwise). To simulate the full attack pipeline, I implemented a function `train`, which is used by both the target model and the shadow models to generate their respective datasets during training.

More specifically, during training, I appended the model's output to `X` at the last epoch. The model output logits represent the predicted probabilities for each class. For each image in the training phase, I appended 1 to `Y`, and for each image in the validation phase, I appended 0.

1.2 Train Shadow Model

(5 points) The steps to prepare the dataset for the attack model are as follows:

1. Generate train and test datasets using the class `custom_MNIST`, and apply `transforms` to normalize the inputs.

2. Create dataloaders for both train and test datasets. Set the parameters `batchsize=64` and `shuffle=True`, as same as those used for the target model.

3. Train 20 shadow models using the SGD optimizer for 40 epochs each.

4. Collect the binary datasets $\{logits_i, bool_i\}_{i=1}^{N}$ for each shadow model and concatenate them. The final attack model's training dataset is composed of 20 such sub-datasets.

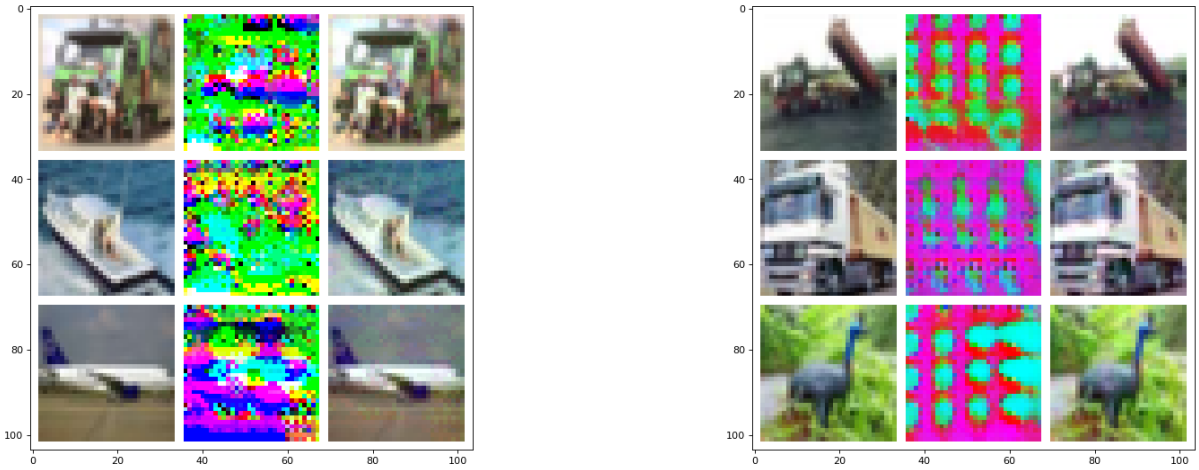
After training, each shadow model achieved around 90% validation accuracy.

1.3 Notice

I train 20 shadow models to reduce overfitting and improve robustness. Although in this homework the architecture of the target model is known, in many real-world scenarios it is not. Hence, we aim to exploit the transferability between models with similar architectures. Improving robustness and avoiding overfitting can significantly enhance the attack success rate.

2 Unlearn Defense

I implemented `Samplewise.ipynb` and `Classwise.ipynb`, which respectively make individual samples and a specific class unlearnable. The final results are shown in Figure 2. The clean accuracy dropped to 17% for the samplewise defense and 10% for the classwise defense.



(a) Examples of samplewise noise

(b) Examples of classwise noise

Figure 2: Perturbations after generating unlearnable data

Both defense approaches follow two main steps, which can be repeated multiple times:

1. Train the model parameters θ by minimizing the classification loss using data $\{x_i + \delta, y_i\}_{i=1}^{N}$.
2. Use the function `min_min_attack` to generate perturbations for a batch of images by returning a list `eta`. After obtaining δ, repeat step 1 to continue training the model.

2.1 Classwise

In the classwise unlearn defense, the shape of the noise is `[10, 3, 32, 32]`. The following are the implementation details:

2.1.1 Train Model Parameters

(2 points) Use the label of each image to retrieve the corresponding noise. Restart iterator if the dataset has been run out.

```
    try:
        (images, labels) = next(data_iter)
    except: # if the data is run out, reset the iterator
        train_idx = 0
        data_iter = iter(clean_train_loader)
        (images, labels) = next(data_iter)
    for i, (image, label) in enumerate(zip(images, labels)):
        # Update noise to images
        images[i] += noise[label]
        train_idx += 1
    images, labels = images.cuda(), labels.cuda()
    base_model.zero_grad()
    optimizer.zero_grad()
    logits = base_model(images)
    loss = criterion(logits, labels)
    loss.backward()
    optimizer.step()
```

2.1.2 Update Noise

(2 points) For each batch, use `min_min_attack` to generate a list `eta` of perturbations. For any class i that appears in the batch, compute the average δ of all samples in that class and update the noise dictionary accordingly. For classes not present in the batch, keep the existing noise values unchanged.

2.1.3 Add Trained Noise to Clean Dataset

(2 points) We generate the unlearnable dataset in two steps:

1. For each data point in the dataset, use its label to retrieve the corresponding noise.
2. Add the noise to the clean image and clip the result within 255.

2.2 Samplewise

(6 points) The key difference between samplewise and classwise defense lies in how the noise is stored and updated. In samplewise defense, the noise dictionary has shape `[50000, 3, 32, 32]`. For each sample, I optimize its own δ in a similar fashion to classwise defense. Below is the detailed code:

```
while condition:
    base_model.train()
    for param in base_model.parameters():
        param.requires_grad = True
    for j in range(0, 10): # 10 steps
        try:
            (images, labels) = next(data_iter)
        except:
            train_idx = 0
            data_iter = iter(clean_train_loader)
            (images, labels) = next(data_iter)
        for i, (image, label) in enumerate(zip(images, labels)):
            # Update noise to images
            images[i] += noise[train_idx]
            train_idx += 1
        images, labels = images.cuda(), labels.cuda()
        base_model.zero_grad()
        optimizer.zero_grad()
        logits = base_model(images)
        loss = criterion(logits, labels)
        loss.backward()
        optimizer.step()

    # Perturbation over entire dataset
    idx = 0
    for param in base_model.parameters():
        param.requires_grad = False
    for i, (images, labels) in tqdm(enumerate(clean_train_loader),
       total=len(clean_train_loader)):
        #    batch
        batch_start_idx, batch_noise = idx, []
        for i, _ in enumerate(images):
            batch_noise.append(noise[idx])
            idx += 1
        batch_noise = torch.stack(batch_noise).cuda()

        # Update class-wise perturbation
        base_model.eval()
        images, labels = images.cuda(), labels.cuda()
        perturb_img, eta = noise_generator.min_min_attack(...)
        for j,delta in enumerate(eta):
            noise[batch_start_idx+j] = delta.clone().detach().cpu()

    eval_idx, total, correct = 0, 0, 0
    for i, (images, labels) in enumerate(clean_train_loader):
        for i, _ in enumerate(images):
            images[i] += noise[eval_idx]
            eval_idx += 1
        images, labels = images.cuda(), labels.cuda()
        with torch.no_grad():
```

```
            logits = base_model(images)
            _, predicted = torch.max(logits.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    acc = correct / total
    print('Accuracy %.2f' % (acc*100))
    if acc > 0.99:
        condition=False
```

The main difference here is that we track both the global input index and the batch index to store and retrieve the corresponding noise for each individual sample.

2.3 Analysis

After training for several epochs, I obtained the noise dictionaries for both classwise and samplewise defenses. A visual summary of the noise patterns is shown in Figure 2. Although the two types of noise appear quite different (classwise noise has a more sparse distribution, while samplewise noise is more concentrated), I believe this is because samplewise defense must encode 50,000 distinct perturbations, and a Gaussian-like pattern may be more expressive. In contrast, classwise perturbations only need to encode 10 classes, so a simpler pattern, such as the one shown in Figure 2b, suffices.

After multiple training runs and sampling different noises, I also observed that the noise patterns varied significantly between different experiments. This variance may be due to differences in training effectiveness and the specific examples used.

Finally, the accuracy on the clean test set dropped to 10% for classwise defense and 17% for samplewise defense. Meanwhile, the accuracy on the unlearnable training dataset remained above 99%, indicating that both defense methods were successfully implemented.