Report: Homework 3

Mao Chuan, 2300013218

May 19, 2025

1 Backdoor Attack

In this part, I implemented three `Backdoor Attack` algorithms: **BadNets, Blend, Clean Label**. The process includes generating the trigger pattern, creating poisoned data (for the clean label attack), and applying the trigger to test samples to evaluate the success of the backdoor attack.

After training, the performance of the trained network exceeds the full-score line. The training logs are stored in the directory: `logs`.

1.1 Trigger Generation

Based on the requirements, trigger generation is categorized into three types:

- **Bottom-right trigger**: Set the bottom-right 3×3 pixels as the trigger pattern using `image[32-3+i, 32-3+j] = pattern[i,j]`. (All pattern transparency values are set to 0.) Then set the corresponding positions in the mask to 1 for pixels originally valued at 0.
- **Checkerboard (4-corner) trigger**: Use a method similar to the bottom-right trigger version. Set the mask to 1 for the trigger pixels.
- **Gaussian noise trigger**: Load the Gaussian image from `cifar_gaussian_noise.png` and use it as the trigger pattern. Set the mask to 1 for all pixels.

1.2 Adding Triggers to the Dataset

1.2.1 Adding Triggers to Poisoned Dataset

In the task `add_trigger_cifar`, which supports only BadNets and Blend attacks, the procedure is as follows: First, filter the dataset to select candidate samples whose labels are not equal to the target class of the backdoor attack. Then, randomly select a subset of these candidates and apply the trigger. After that, change their labels to the attack target label. In the codebase, the poison rate is set as `poison_rate=0.05`. If the poison rate is too high, the clean accuracy may degrade rapidly. The method to apply the trigger is:

$$poisoned\_img = (1 - \text{mask}) \times \text{image} + \text{mask} \times ((1 - \alpha) \times \text{image} + \alpha \times \text{pattern})$$

In specific attack instances:

1. For BadNets and Clean Label attacks, $\alpha = 1.0$, which means replacing the corresponding image pixels with the pattern (e.g., set to 0 or 255). The original paper also explores smaller values of α, which make the attack more stealthy.

2. For the Blend attack, $\alpha = 0.2$, which mixes the image and Gaussian noise, with weights $1 - \alpha$ and α, respectively.

1.2.2 Adding Triggers to the Test Dataset

To construct the test dataset for evaluating the poisoned accuracy, I filtered out images whose labels are already equal to the target class of the attack. This step is important to ensure that the effect of the trigger can be properly measured.

1.3 Generating Clean Label Poisoned Data

To obtain poisoned data for the clean label attack, I implemented the **PGD (Projected Gradient Descent)** algorithm to generate adversarial samples. In the outer loop (repeated `restart` times), the algorithm selects the perturbation δ that leads to the maximum loss. Within each outer iteration, the inner loop (run for `max_attack_iters` iterations) performs the following PGD steps:

1. Randomly initialize the adversarial sample within the allowed perturbation range:

$$x_{adv}^{(0)} = x + \text{random}(-\epsilon, \epsilon)$$

   where $x_{adv}^{(k)}$ denotes the adversarial sample at the k-th iteration.

2. Iteratively update x_{adv} for T steps:

   - Compute the gradient of the cross-entropy loss:

$$g_t = \nabla_{x_{adv}} \mathcal{L}(f(x_{adv}^{(t)}), y)$$

   - Update the adversarial sample using the sign of the gradient:

$$x_{adv}^{(t+1)} = x_{adv}^{(t)} + \alpha \cdot \text{sign}(g_t)$$

     where α is the step size, typically set to $\frac{\epsilon}{T}$.

   - Project the updated sample back into the valid perturbation range:

$$x_{adv}^{(t+1)} = \text{Clip}_{x,\epsilon}(x_{adv}^{(t+1)})$$

Important implementation notes:

1. Unlike the standard PGD attack, this implementation breaks out of the inner loop early if the benign model misclassifies the adversarial sample. This saves computation while still achieving the goal of clean label poisoning.

2. The original PGD paper uses gradients with respect to x_{adv} and updates x_{adv} at each step.

Table 1: Training Results

Task	Poison Accuracy	Clean Accuracy
BadNetss	100%	91.46% ($\geq$ 91%)
Blend	100%	92.79% ($\geq$ 91%)
Clean Label	89.24% ($\geq$ 80%)	93.22% ($\geq$ 91%)

1.4 Training Results

The training results for all three backdoor attack methods reach the full score threshold. The detailed performance on the 50th (final) epoch is shown in Table 1.

2 Backdoor Defense

In this section, I implement a backdoor defense mechanism using **Adversarial Neuron Pruning (ANP)** without access to poisoned data.

2.1 Code Implementation: Training Masks

The paper *Adversarial Neuron Pruning Purifies Backdoored Deep Models* proposes pruning neurons associated with backdoor triggers to defend against backdoor attacks. The key idea is to identify and remove neurons that are sensitive to adversarial perturbations, as these neurons are likely to be responsible for the backdoor behavior. The implementation involves the following steps:

1. **Identifying Sensitive Neurons**: Neurons that 'memorize' triggers can significantly influence classification results. Perturbations to their parameters may lead to larger losses. To identify such neurons, we generate perturbations for the network parameters. In each inner iteration, we use an optimizer to find $\delta, \xi \in [-\epsilon, \epsilon]^n$ that maximize the loss:

$$\max_{\delta,\xi\in[-\epsilon,\epsilon]^n} \mathcal{L}((1+\delta)\odot w + (1+\xi)\odot b)$$

   In the code, I use `-CrossEntropyLoss(output, label)` as the loss function. After each update of δ and ξ, I clip them to maintain the perturbation within the allowed range.

2. **Training the Mask** m: After obtaining δ_k and ξ_k from the previous step, we aim to find a mask $m \in [0,1]^n$ by optimizing the following objective:

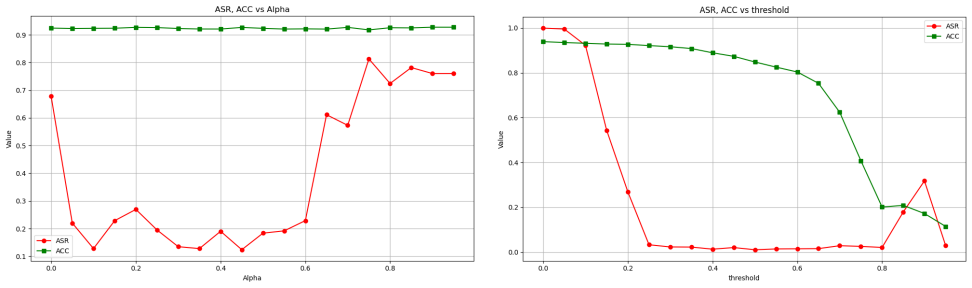
$$\min_{m\in[0,1]^n} \alpha \cdot \text{acc\_loss} + (1-\alpha)\cdot \text{rob\_loss}$$

   where acc_loss $= \mathcal{L}(m \odot w + b)$ and rob_loss $= \mathcal{L}\big((m+\delta_k)\odot w + (1+\xi_k)\odot b\big)$. We do not assign weights to b and never delete b, as this may cause performance degradation.

In my implementation, I first include noise (δ_k, ξ_k) to calculate the robustness loss, then exclude noise to calculate the accuracy loss. The sum of these two terms is used to update m through optimization.

2.2 Hyperparameter Tuning

I implemented `run_exp.py` to automate command-line inputs and program execution. The script parses key outputs, saving results such as **PoisonACC** and **CleanACC** to a JSONL file. Finally, I use Matplotlib to plot curves illustrating the influence of different parameters on accuracy. The results are shown in Figure 1.



(a) Varying α with `threshold`=0.2 (b) Varying threshold with $\alpha = 0.2$

Figure 1: Hyperparameter tuning for α and threshold

2.2.1 Choosing α

Due to the large two-dimensional parameter space, the original paper sets the threshold to zero and searches for an optimal α. Here, we set `threshold`=0.2 and run `generate_mask.py` with α ranging from 0.0 to 1.0 in steps of 0.05. I define the score as:

$$\texttt{score} = (0.05 - \texttt{PoisonACC}) + (\texttt{CleanACC} - 0.92)$$

In Figure 1a, we observe the influence of α on accuracy and attack success rate (ASR). With an appropriately set threshold, the network architecture remains intact, maintaining accuracy above 90%, indicating good performance on clean data. In the optimization equation above, a larger α places more emphasis on accuracy. When $\alpha > 0.6$, robustness is neglected, leading to an increased ASR. The mask m is not sufficiently small, and the 'bad' neurons are not pruned effectively, weakening the defense.

2.2.2 Choosing Threshold

I select subsets of α where the score exceeds -0.25. For each α, I vary the threshold from 0.0 to 1.0 in steps of 0.05. I find that certain parameter combinations (e.g., $\alpha = 0.2$, `threshold`=0.25) achieve full scores. A screenshot supporting this result is shown in Figure 2.

No.	Layer Name	Neuron Idx	Mask	PoisonLoss	PoisonACC	CleanLoss	CleanACC
0	None	None	0.0042	1.0000	0.2491	0.9389	
196.00	layer3.1.bn1	153 0.25	3.6408	0.0330	0.2727	0.9213	

Figure 2: Test output with $\alpha = 0.2$, `threshold`=0.25

During this process, I record and plot the influence of the threshold on PoisonACC and CleanACC in Figure 1b. We observe that the ASR decreases sharply as the threshold increases from 0.1 to 0.25, while the clean accuracy remains above 90%. This indicates that the defense method is effective, enhancing robustness without compromising accuracy. However, when the threshold exceeds 0.6, accuracy declines, suggesting that the network architecture has been compromised.

More specifically, after training weights for each neuron in the network, we obtain a mask m for each neuron. If the mask is effective, neurons that 'memorize' the trigger typically have smaller m values. Therefore, when the threshold is in the range of 0.1 to 0.25, most of these neurons are pruned, leading to a decreased ASR. However, as the threshold increases further, some beneficial neurons that capture important features for different image classes are also removed, resulting in decreased accuracy.