Secure AI Systems: Red and Blue Teaming an MNIST Classifier

Aryan Gupta
IIIT Hyderabad

September 26, 2025

Abstract

This report documents the design, attacks (red teaming), defenses (blue teaming), and evaluation of a CNN classifier trained on the MNIST handwritten digits dataset. Included are the training details, performance metrics on clean and adversarial data, confusion matrices and discussion of threat modelling and static analysis (SAST).

GitHub repo (public)
Kindly please note that gen ai (LLMs) were used to fix and format the report and code

1 Introduction

This assignment explores secure AI systems through a red/blue team exercise on a simple CNN for MNIST digit classification. We train a baseline model, generate adversarial examples (FGSM) and poisoned samples, measure the impact on model performance, then perform adversarial training to improve robustness.

2 Dataset and Preprocessing

We use the standard MNIST dataset (28x28 grayscale images). For model training we normalize inputs using mean 0.1307 and std 0.3081. For visualization and some adversarial generation steps we use unnormalized pixel ranges $[0, 1]$.

3 Model Architecture and Training

The CNN architecture used across baseline and robust models:

- Conv2d(1, 32, kernel_size=3, padding=1) + ReLU + MaxPool2d(2)
- Conv2d(32, 64, kernel_size=3, padding=1) + ReLU + MaxPool2d(2)
- Fully connected 7*7*64 $\rightarrow$ 128 + ReLU
- FC 128 $\rightarrow$ 10 (output logits)

Optimizer: Adam, initial LR = 0.001. Loss: CrossEntropyLoss. Baseline training: 5 epochs. Adversarial training uses FGSM samples generated with ART, poisoned and mixed with clean data for re training.

4 Threat Modeling (Red Team)

We consider the following adversarial capabilities and goals:

- **Data poisoning**: Attacker can inject a small number of labeled training examples with a visible trigger (4x4 colored corner square) to cause model misbehavior for triggered inputs.
- **Evasion via adversarial perturbation**: Attacker crafts near imperceptible perturbations (FGSM/PGD) at inference time to cause misclassification.
- **Goals**: reduce model accuracy on target inputs, create targeted misclassification, or degrade system reliability.

Adversary model assumptions:

- Surrogate access used for FGSM generation (ART's FGSM) link.
- For poisoning the attacker is able to add ≈ 100 poisoned samples of digit '7' in the training corpus.

5 Threat Modeling

This section outlines the potential security threats to the MNIST handwritten digit classifier application using the STRIDE framework. Each category describes threats, scenarios, impacts, and mitigations.

5.1 Spoofing

Threat: An attacker could spoof the identity of the data source, providing malicious or poisoned data to the training pipeline.
Scenario: An attacker intercepts the dataset download or local data loading process and injects backdoor images (e.g., all images with a specific pixel pattern are classified as '9').
Impact: The trained model becomes unreliable and contains a hidden backdoor that the attacker can exploit.
Mitigation: Use dataset hashing and digital signatures to verify dataset integrity. Implement access controls on storage locations. Employ reproducible dataset pipelines (e.g., checksums in CI/CD) to ensure data consistency across environments.

5.2 Tampering

Threat: An attacker could tamper with the model file or the training script.
Scenario 1 (Model Tampering): After training, the saved model file (`mnist_cnn_baseline.pth`) could be replaced with a malicious model that always predicts a specific digit.
Scenario 2 (Code Tampering): An attacker could modify the training script (`1_train_evaluate_baseline.p`) to log sensitive data, alter the model, or introduce subtle bugs.
Impact: Model integrity is compromised, leading to incorrect predictions or system compromise.
Mitigation: Use file integrity monitoring (e.g., hashing). Store models in access-controlled, versioned repositories. Require code reviews and signed commits for training scripts. Use secure build pipelines with artifact signing to prevent tampering in CI/CD environments.

5.3 Repudiation

Threat: A user or system denies having performed an action.
Scenario: In production, a user could deny submitting a specific digit image that led to an incorrect or disputed outcome.
Impact: Low risk for a toy classifier, but critical in systems where classifications trigger financial or safety actions. Disputes may be unresolvable without verifiable logs.
Mitigation: Implement tamper proof logging of prediction requests, including input hashes, model outputs, user identity, and timestamps. Consider audit trails backed by append only storage (e.g., blockchain or write once logs) for high assurance environments.

5.4 Information Disclosure

Threat: An attacker could extract sensitive information about the training data or model architecture.
Scenario 1 (Model Stealing): An attacker with API access trains a copycat model by querying the classifier repeatedly.
Scenario 2 (Membership Inference): An attacker determines if a specific user's handwritten digit was part of the training set, violating privacy.
Impact: Intellectual property loss, privacy violations.
Mitigation: Limit API query rates. Use differential privacy techniques during training. Employ model watermarking to detect stolen models. Apply output perturbation or confidence masking to reduce information leakage.

5.5 Denial of Service (DoS)

Threat: An attacker prevents legitimate users from accessing the model.
Scenario: An attacker floods the API with prediction requests, exhausting CPU/GPU resources.
Impact: Service becomes unresponsive or unavailable.
Mitigation: Enforce rate limiting, authentication, and resource monitoring. Use scalable infrastructure with auto scaling. Deploy Web Application Firewalls (WAF) and anomaly detection to identify abusive patterns.

5.6 Elevation of Privilege

Threat: An attacker exploits vulnerabilities in the application code or dependencies to gain unauthorized access.
Scenario: A vulnerability in PyTorch, NumPy, or insecure use of `pickle`/`torch.load` could allow arbitrary code execution on the host system.
Impact: Full compromise of the server, leading to data exfiltration, system tampering, or lateral movement attacks.
Mitigation: Apply the principle of least privilege. Keep dependencies updated and run vulnerability scans. Use sandboxing/containerization (e.g., Docker) for isolation. Avoid insecure deserialization methods. Enforce mandatory access controls (e.g., SELinux, AppArmor) to restrict process capabilities. Perform dependency pinning and SBOM (Software Bill of Materials) analysis to detect supply chain risks.

6 Bandit B614: static analysis security testing (SAST)

Issue Identified
Bandit flagged the use of `torch.load()` in two files (`adv_gen.py` line 42 and `eval_attack.py`

line 60). The problem is that `torch.load()` internally relies on Python's `pickle` module, which can execute arbitrary code during deserialization. If the model file path (`MODEL_PATH`) were ever influenced by untrusted input, this could lead to remote code execution. Bandit assigned this issue **Severity: Medium** and **Confidence: High**, mapping it to **CWE-502 (Deserialization of Untrusted Data)**.

How Bandit Detected It

Bandit uses rule **B614 (pytorch_load)** to scan for unsafe usage of `torch.load()`. Any occurrence of this function without safeguards is flagged because of its reliance on `pickle`. Both occurrences in our codebase were caught automatically.

Fix Implemented

We updated the model loading logic to use PyTorch's safer mechanism available in version $\geq$ 2.0 by specifying `weights_only=True`. This ensures only raw tensor weights are deserialized, not arbitrary Python objects. In addition, we confirmed that our training pipeline saves only `state_dict`s instead of full model objects.

Before (insecure):

```
model = MNIST_CNN()
model.load_state_dict(torch.load(MODEL_PATH, map_location=DEVICE))
```

After (secure):

```
state_dict = torch.load(MODEL_PATH, map_location=DEVICE, weights_only=True)
model = MNIST_CNN()
model.load_state_dict(state_dict)
```

Result

After applying this fix, Bandit was re-run and reported **no issues identified**. The risk of code execution through maliciously crafted model files has been mitigated, while maintaining full model functionality.

7 Attack Implementation (Red Team)

Two attack modalities were implemented:

7.1 Data Poisoning

We modified $\approx$ 100 training images of digit '7' by adding a small colored 4×4 square in the corner. The poisoned data was retained with original labels (target = 7) — this is a classic trigger based poisoning.

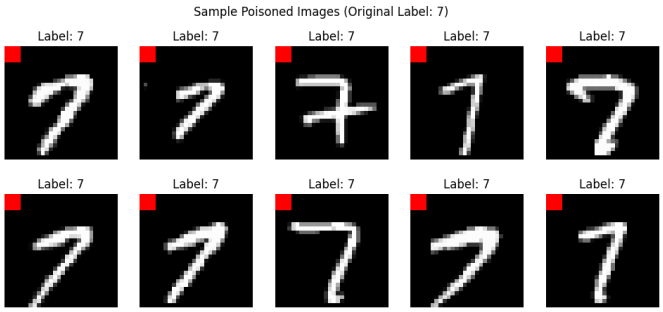


Figure 2: Sample poisoned images with 4x4 colored trigger in the corner (original label: 7).

Metrics:

Total lines of code: 351
Total lines skipped (#nosec): 0

pytorch_load: Use of unsafe PyTorch load
Test ID: B614
Severity: MEDIUM
Confidence: HIGH
CWE: CWE-502
File: ./adv_gen.py
Line number: 42
More info: https://bandit.readthedocs.io/en/1.8.6/plugins/b614_pytorch_load.html

```
41      model = MNIST_CNN()
42      model.load_state_dict(torch.load(MODEL_PATH, map_location=DEVICE))
43      model.to(DEVICE).eval()
```

pytorch_load: Use of unsafe PyTorch load
Test ID: B614
Severity: MEDIUM
Confidence: HIGH
CWE: CWE-502
File: ./eval_attack.py
Line number: 60
More info: https://bandit.readthedocs.io/en/1.8.6/plugins/b614_pytorch_load.html

```
59      model = MNIST_CNN().to(DEVICE)
60      model.load_state_dict(torch.load(MODEL_PATH, map_location=DEVICE))
61
```

Figure 1: SAST flag

7.2 Adversarial Evasion (FGSM)

We generated adversarial test samples using ART's `FastGradientMethod` (FGSM) with $\epsilon = 0.2$ and saved the adversarial test set.

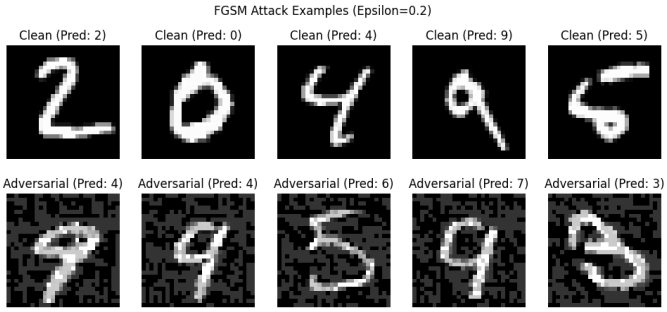


Figure 3: FGSM attack examples: top row clean; bottom row adversarial (predicted labels).

8 Evaluation Metrics and Results

Below are the performance numbers collected from your experiments.

8.1 Baseline Model (Clean Test Set)

Metric	Value
Accuracy (clean test set)	98.91%
Average Loss (clean test set)	0.0411

Table 1: Baseline model performance on the clean test set.

8.2 Baseline Model on Adversarial Data (Red Team)

Metric	Value
Accuracy on adversarial samples	54.72%
Average loss on adversarial samples	1.7762

Table 2: Baseline model evaluated on adversarial test set.

8.3 Robust Model (Blue Team)

Metric	Value
Accuracy on Clean Test Set	98.76%
Average Loss on Clean Test Set	0.0424
Accuracy on Adversarial Test Set	89.94%
Average Loss on Adversarial Test Set	0.3371

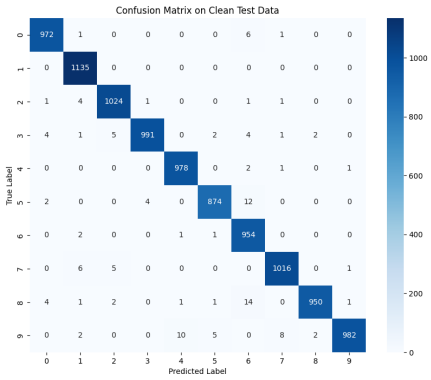
Table 3: Robust model performance after adversarial training.

8.4 Inference Time

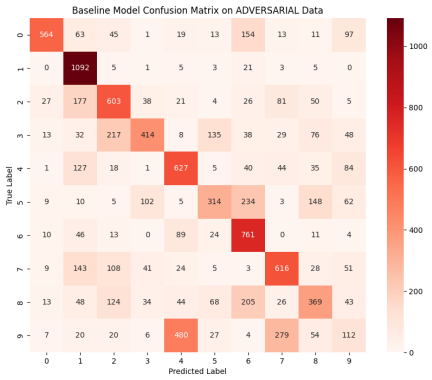
Metric	Value
Batch size	1000
Average inference time per batch	0.020487 seconds
Average inference time per image	0.000020 seconds

Table 4: Inference time measurements.

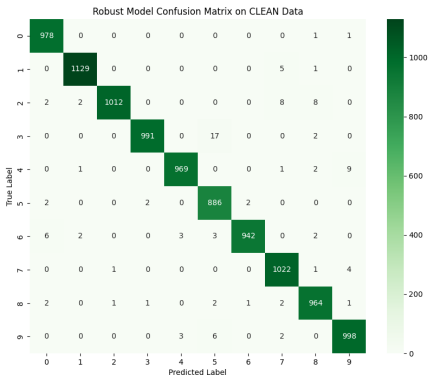
9 Confusion Matrices



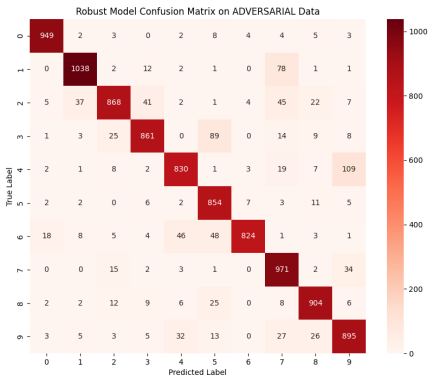
(a) Baseline model on clean test set



(b) Baseline model on adversarial test set



(c) Robust model on clean test set



(d) Robust model on adversarial test set

Figure 4: Confusion matrices.

10 Blue Team Actions & Mitigations

- **Adversarial training:** we mixed FGSM generated adversarial examples with clean training data and re trained the model. This improved adversarial accuracy from 54.72% → 89.94%.
- **Data sanitation:** detect corner triggered poisoned samples (small colored corner) using simple heuristics (color detection) or train a small binary detector to filter out suspicious training samples (out od training data distribution).

- **Input pre processing:** techniques like JPEG compression or randomized smoothing can reduce some adversarial effects.
- **Model hardening:** use adversarial training with stronger attacks (PGD), ensemble methods .etc.
- **SAST + CI checks:** integrate Bandit/Semgrep checks into CI to flag unsafe loading, insecure deserialization, and suspicious file operations.
- **Monitoring:** log anomaly statistics, prediction confidence shifts, and drift detectors in production to detect sudden increases in misclassification rates.

11 Discussion

The baseline model had excellent clean accuracy (98.8%) but was highly vulnerable to FGSM adversarial examples (accuracy dropped to 54.72%). Adversarial training significantly improved robustness (adversarial accuracy to 89.94%) while retaining clean accuracy. The data poisoning trigger used is easily spotted visually but is a realistic supply chain style threat if training data is not validated. Combining detection, sanitization, and adversarial training leads to a more resilient pipeline.

12 Conclusion

This exercise demonstrates how simple attacks (FGSM and trigger based poisoning) can severely degrade model performance, and how blue team defenses such as adversarial training and data sanitation can restore robustness. For production systems, combine multiple defenses, SAST/S-DLC hygiene, and runtime monitoring.

Appendix

- `train_eval_baseline.py` – trains baseline CNN, evaluates on clean test set, saves `baseline_results/mni` and `confusion_matrix_baseline.png`.
- `adv_gen.py` – uses ART FGSM to produce `attack_data/adversarial_test_set.npz` and `fgsm_attack_visualization.png`.
- `data_poisoning.py` – creates poisoned images and poisons the dataset.
- `adv_train.py` – generates adversarial training samples, re-trains model (robust), saves `adversarial_training_results/mnist_cnn_robust.pth` and confusion matrices.
- `eval_attack.py` – evaluates baseline on adversarial set and writes `attack_evaluation_results/adversa` and confusion matrix image.
- `run_time_infernece.py` – measures inference time and writes `attack_evaluation_results/inference.`