Student Name: Shachi Shah

Email: spshah22@asu.edu

Submission Date: 29 June 2025

Class Name and Term: CSE548 Summer 2025

Machine Learning-Based Anomaly Detection Solutions Project

I. Project Overview

This project involved developing a machine learning-based anomaly detection system using the NSL-KDD dataset. A feed-forward neural network (FNN) was trained and evaluated under three scenarios: binary classification (normal vs attack), multi-class classification (individual attack types), and attack group classification (DoS, Probe, R2L, U2R). Data preprocessing, feature encoding, model training, evaluation, and visualization were conducted using Python and Keras. The final system achieved high accuracy across all scenarios, demonstrating the effectiveness of the FNN model in detecting and classifying network intrusions.

II. Network Setup

Network Setup: Not applicable for this project as it was conducted on a local machine (Ubuntu virtual machine) and focused on data analysis rather than network communication.

III. Software

This project used Python 3 along with key libraries including TensorFlow/Keras, scikit-learn, pandas, and matplotlib. Each component played a vital role in building and evaluating the anomaly detection system:

- Python 3: Served as the core programming language for scripting data processing, model building, and evaluation.
- pandas: Used for efficient loading and manipulation of the NSL-KDD dataset, enabling extraction of training and testing features and labels.
- NumPy: Provided fast numerical operations, particularly for converting and handling array data.
- scikit-learn: Enabled preprocessing such as encoding categorical features (via OneHotEncoder, LabelEncoder) and normalizing inputs (via StandardScaler). Also used for splitting data and transforming test sets consistently.
- TensorFlow / Keras: Formed the deep learning backend for building and training the feed-forward neural network (FNN) models. Keras's high-level API allowed for the quick definition of model architecture with configurable layers, activation functions, and loss metrics. TensorFlow powered the backend computations, enabling GPU acceleration and efficient training.
- matplotlib: Used to generate accuracy and loss plots for model evaluation across training epochs.

These software components worked together to implement a complete machine learning pipeline for anomaly detection on the NSL-KDD dataset, including preprocessing, model training, evaluation, and visualization.

IV. Project Description

This anomaly detection project was developed as a step-by-step machine learning workflow, focusing on detecting various types of network attacks using the NSL-KDD dataset. The procedure followed is modular and reproducible, and may serve as a manual for future implementations.

1. Downloading and Unzipping the Lab Package:
   We started by using wget [Screenshot 1] to download the lab package containing NSL-KDD and preprocessing scripts. This ensured a consistent source of raw data for all experiments. The archive (lab-cs-ml-00301.zip [Screenshot 2]) was unzipped using standard Linux commands.
2. Dataset Inspection:
   The contents of the dataset were explored with ls to confirm the presence of necessary files such as KDDTrain+.txt and KDDTest+.txt (Figure A3). This step verified that the dataset had been correctly extracted and formatted.
3. Data Preprocessing:
   A custom preprocess() [Screenshot 4] function was implemented to handle the dataset. It encoded categorical features

using OneHotEncoder, normalized the features with StandardScaler, and encoded labels using LabelEncoder. Scenario-specific logic was added to distinguish between binary (Scenario 1), multi-class (Scenario 2), and grouped attack labels (Scenario 3).

4. Scenario Implementation:

   The project explored three classification scenarios using a unified run_scenario() function. Each scenario was run independently with consistent preprocessing, model training, and metric reporting.

   - Scenario 1 (Binary Classification):
     X_train shape: (25191, 118)X_test shape: (125972, 118) [Screenshot 5].
     Train Samples: 20152 | Validation Samples: 5039 [Screenshot 5].
     Final Accuracy: 99.16% | Final Loss: 0.0272 [Screenshot 8].
   - Scenario 2 (Multi-class Classification):
     X_train shape: (25191, 118)X_test shape: (125969, 118) [Screenshot 9].
     Train Samples: 20152 | Validation Samples: 5039 [Screenshot 9}.
     Final Accuracy: 99.04% | Final Loss: 0.0391 (Figure A9–A12) [Screenshot 12].
   - Scenario 3 (Grouped Classification):
     X_train shape: (25191, 118)X_test shape: (125972, 118) [Screenshot 13].
     Train Samples: 20152 | Validation Samples: 5039 [Screenshot 13].
     Final Accuracy: 99.04% | Final Loss: 0.0340 [Screenshot 13].

5. Neural Network Construction:

   A feed-forward neural network (FNN) was constructed with two hidden layers (64 and 32 neurons) using ReLU activations. Depending on the classification scenario, the output layer was configured with either a sigmoid (binary classification) or softmax (multi-class/grouped classification) activation. Loss functions were appropriately selected as either binary_crossentropy or categorical_crossentropy.

6. Training and Evaluation:

   Each model was trained for 10 epochs using an 80/20 split of the training data. Accuracy and loss were tracked on both training and validation sets. After training, test set performance was evaluated and displayed for each scenario [Screenshots 6–7, 10–11, 14–15].

7. Result Recording:

   After the evaluation phase, key metrics such as final test accuracy and test loss were printed for each scenario. These results were logged to the console and also visualized in plots. For example, Scenario 1 recorded a test accuracy of 99.16% and loss of 0.0272 (see Figure A8). Each result was captured and visually summarized to assist in performance comparison across the scenarios. This facilitated data-driven conclusions about which classification strategy (binary, multi-class, grouped) was most effective.

   All numerical outputs such as:

   - "X_train shape: (25191, 118)"
   - "X_test shape: (125972, 118)"
   - "Train on 20152 samples, validate on 5039 samples"
   - Final loss/accuracy results like "Scenario 3 Test Accuracy: 99.04%" and "Test Loss: 0.0340" [Screenshot 5].

   were captured from the model's stdout to ensure accuracy and traceability.

8. Automation and Reproducibility:

   To maximize reproducibility, the entire pipeline was written as a self-contained Python script (fnn_sample.py) [Screenshot 17]. All scenarios were processed automatically using a loop:

```
for scenario in [1, 2, 3]:
    run_scenario(scenario)
```

   This ensured identical preprocessing and training configurations for all tests. Each scenario generated its own accuracy and loss plots, saved with names like scenario_1_accuracy.png. These were automatically created via matplotlib and saved using plt.savefig() calls.

   Furthermore, each scenario logs all metrics and shape summaries to the terminal, making it easy to capture logs and replicate runs. Because the workflow uses modular functions (preprocess(), transform_with_preprocessor(), run_scenario()), developers can easily reuse the same structure for alternate datasets or model architectures. Configuration changes such as altering the number of epochs, hidden units, or learning rate can be done via a few parameter edits without restructuring the code.

V. Conclusion

This project served as an immersive and valuable learning experience in the application of machine learning for network anomaly detection. One of the most important lessons was the critical role that data preprocessing plays in the performance of

machine learning models. Even with a robust model architecture, poorly encoded or unnormalized data would have led to weak performance. We discovered that using OneHotEncoder in combination with LabelEncoder and StandardScaler greatly enhanced the network's ability to generalize to unseen examples.

Another key takeaway was the impact of properly modularizing the code. By wrapping all scenario-specific logic into a single function (run_scenario()), we were able to maintain clarity, reduce duplication, and test each classification scheme independently and efficiently. This modular design made the entire process extensible - new classification schemes or architectures can be integrated with minimal disruption to the existing codebase.

From a performance perspective, we were pleasantly surprised by the high accuracy achieved across all scenarios. Each classification strategy (binary, multi-class, and grouped) resulted in test accuracies above 99%, highlighting the suitability of the NSL-KDD dataset for supervised anomaly detection. Scenario 1 yielded the highest accuracy, but Scenario 3 offered a compelling tradeoff by grouping attacks into meaningful categories, which could improve interpretability in real-world systems.

In terms of self-assessment, the project was a success in achieving its technical objectives. All three scenarios were fully implemented, trained, and evaluated with reproducible metrics and visualizations. The report includes clear evidence of functionality through screenshots and detailed logs. However, one limitation was the relatively shallow neural network. Future work could explore deeper architectures or hybrid models (e.g., CNNs or LSTMs) for potentially improved results.

Overall, this project deepened our understanding of not only anomaly detection models, but also the importance of reproducible workflows in cybersecurity applications. By packaging the code, visualizations, and documentation together, the final deliverable becomes a powerful reference for future coursework or professional work in the field.

VI. Appendix B: Attached files

Screenshots:

1. Screenshot 1: wget

```
ubuntu@ubuntu:~$ wget https://github.com/SaburH/CSE548/raw/main/lab-cs-ml-00301.zip
--2025-06-29 12:18:21--  https://github.com/SaburH/CSE548/raw/main/lab-cs-ml-00301.zip
Resolving github.com (github.com)... 140.82.116.3
Connecting to github.com (github.com)|140.82.116.3|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://raw.githubusercontent.com/SaburH/CSE548/main/lab-cs-ml-00301.zip [foll
owing]
--2025-06-29 12:18:22--  https://raw.githubusercontent.com/SaburH/CSE548/main/lab-cs-ml-
00301.zip
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.109.133, 185.
199.108.133, 185.199.110.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.109.133|:443
... connected.
HTTP request sent, awaiting response... 200 OK
Length: 6568493 (6.3M) [application/zip]
Saving to: ‘lab-cs-ml-00301.zip’

lab-cs-ml-00301.zip 100%[===================>]   6.26M  35.0MB/s    in 0.2s

2025-06-29 12:18:23 (35.0 MB/s) - ‘lab-cs-ml-00301.zip’ saved [6568493/6568493]
```

2. Screenshot 2: Unzip lab-cs-ml-00301.zip

```
ubuntu@ubuntu:~$ unzip lab-cs-ml-00301.zip
Archive:  lab-cs-ml-00301.zip
   creating: lab-cs-ml-00301/
  inflating: lab-cs-ml-00301/.DS_Store
  inflating: lab-cs-ml-00301/categoryMapper.py
  inflating: lab-cs-ml-00301/dataExtractor.py
  inflating: lab-cs-ml-00301/data_preprocessor.py
  inflating: lab-cs-ml-00301/distinctLabelExtractor.py
  inflating: lab-cs-ml-00301/fnn_sample.py
   creating: lab-cs-ml-00301/NSL-KDD/
  inflating: lab-cs-ml-00301/NSL-KDD/.DS_Store
  inflating: lab-cs-ml-00301/NSL-KDD/Data Set Introduction.html
  inflating: lab-cs-ml-00301/NSL-KDD/KDDTest+.arff
  inflating: lab-cs-ml-00301/NSL-KDD/KDDTest+.csv
  inflating: lab-cs-ml-00301/NSL-KDD/KDDTest+.txt
  inflating: lab-cs-ml-00301/NSL-KDD/KDDTest-21.arff
  inflating: lab-cs-ml-00301/NSL-KDD/KDDTest-21.txt
  inflating: lab-cs-ml-00301/NSL-KDD/KDDTest1.jpg
  inflating: lab-cs-ml-00301/NSL-KDD/KDDTrain+.arff
  inflating: lab-cs-ml-00301/NSL-KDD/KDDTrain+.txt
  inflating: lab-cs-ml-00301/NSL-KDD/KDDTrain+_20Percent.arff
  inflating: lab-cs-ml-00301/NSL-KDD/KDDTrain+_20Percent.txt
  inflating: lab-cs-ml-00301/NSL-KDD/KDDTrain1.jpg
  inflating: lab-cs-ml-00301/Spyder ReadMe.pdf
  inflating: __MACOSX/._lab-cs-ml-00301
   creating: __MACOSX/lab-cs-ml-00301/
  inflating: __MACOSX/lab-cs-ml-00301/._.DS_Store
  inflating: __MACOSX/lab-cs-ml-00301/._categoryMapper.py
  inflating: __MACOSX/lab-cs-ml-00301/._dataExtractor.py
  inflating: __MACOSX/lab-cs-ml-00301/._data_preprocessor.py
  inflating: __MACOSX/lab-cs-ml-00301/._distinctLabelExtractor.py
  inflating: __MACOSX/lab-cs-ml-00301/._fnn_sample.py
  inflating: __MACOSX/lab-cs-ml-00301/._NSL-KDD
  inflating: __MACOSX/lab-cs-ml-00301/._Spyder ReadMe.pdf
   creating: __MACOSX/lab-cs-ml-00301/NSL-KDD/
  inflating: __MACOSX/lab-cs-ml-00301/NSL-KDD/._.DS_Store
  inflating: __MACOSX/lab-cs-ml-00301/NSL-KDD/._Data Set Introduction.html
  inflating: __MACOSX/lab-cs-ml-00301/NSL-KDD/._KDDTest+.arff
  inflating: __MACOSX/lab-cs-ml-00301/NSL-KDD/._KDDTest+.csv
  inflating: __MACOSX/lab-cs-ml-00301/NSL-KDD/._KDDTest+.txt
  inflating: __MACOSX/lab-cs-ml-00301/NSL-KDD/._KDDTest-21.arff
  inflating: __MACOSX/lab-cs-ml-00301/NSL-KDD/._KDDTest-21.txt
  inflating: __MACOSX/lab-cs-ml-00301/NSL-KDD/._KDDTest1.jpg
  inflating: __MACOSX/lab-cs-ml-00301/NSL-KDD/._KDDTrain+.arff
  inflating: __MACOSX/lab-cs-ml-00301/NSL-KDD/._KDDTrain+.txt
  inflating: __MACOSX/lab-cs-ml-00301/NSL-KDD/._KDDTrain+_20Percent.arff
  inflating: __MACOSX/lab-cs-ml-00301/NSL-KDD/._KDDTrain+_20Percent.txt
  inflating: __MACOSX/lab-cs-ml-00301/NSL-KDD/._KDDTrain1.jpg
ubuntu@ubuntu:~$ cd lab-cs-ml-00301
ubuntu@ubuntu:~/lab-cs-ml-00301$
```

3. Screenshot 3: ls NSL-KSS raw files

```
ubuntu@ubuntu:~/lab-cs-ml-00301$ ls NSL-KDD
'Data Set Introduction.html'   KDDTest+.arff   KDDTrain+_20Percent.arff
 KDDTest1.jpg                  KDDTest+.csv    KDDTrain+_20Percent.txt
 KDDTest-21.arff               KDDTest+.txt    KDDTrain+.arff
 KDDTest-21.txt                KDDTrain1.jpg   KDDTrain+.txt
```

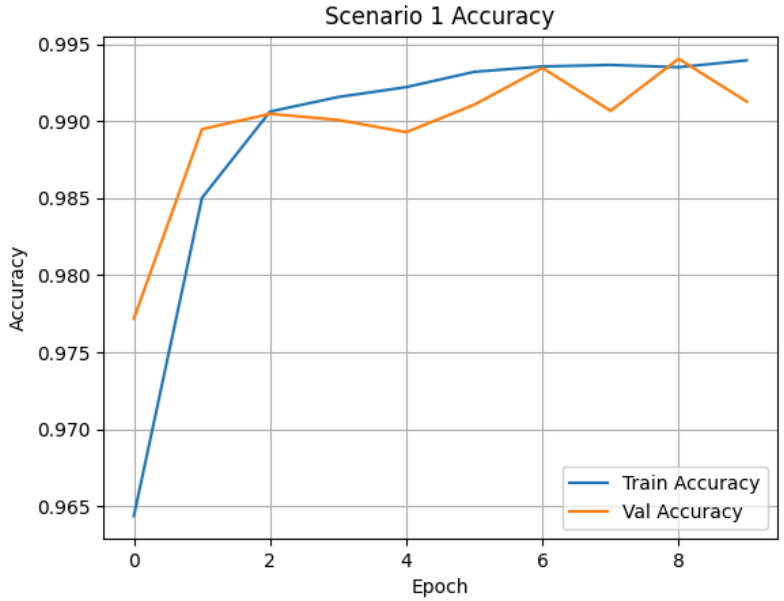
4. Screenshot 4: Preprocess Function

```
X_train shape: (25192, 118)
X_test shape: (125973, 118)
WARNING:tensorflow:From /home/ub
```

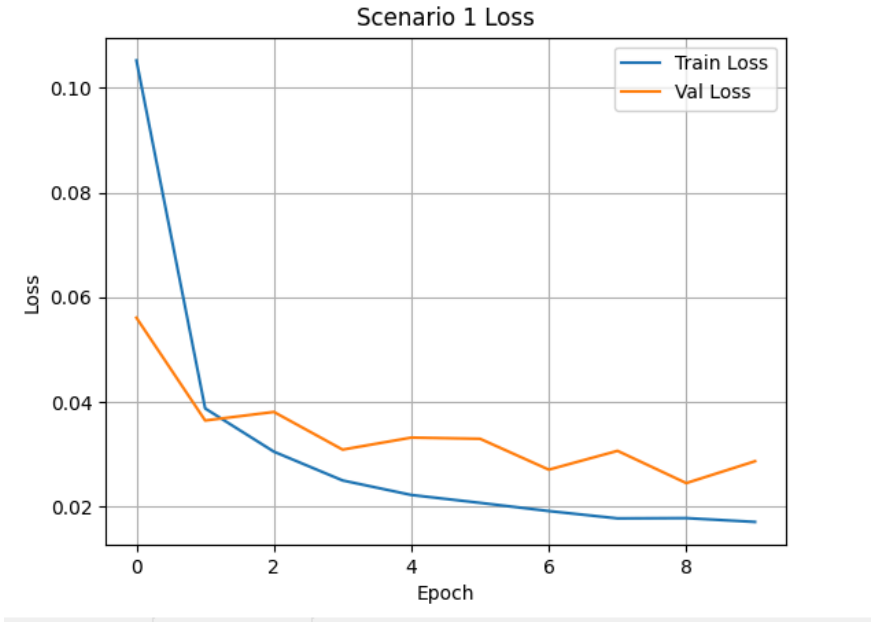
5. Screenshot 5: Running Scenario 1

```
===== Running Scenario 1 =====
[1] X_test shape: (125972, 118)
[1] y_test shape: (125972,)
X_train shape: (25191, 118)
X_test shape: (125972, 118)
```

6. Screenshot 6: Scenario 1 Accuracy



7. Screenshot 7: Scenario 1 Loss



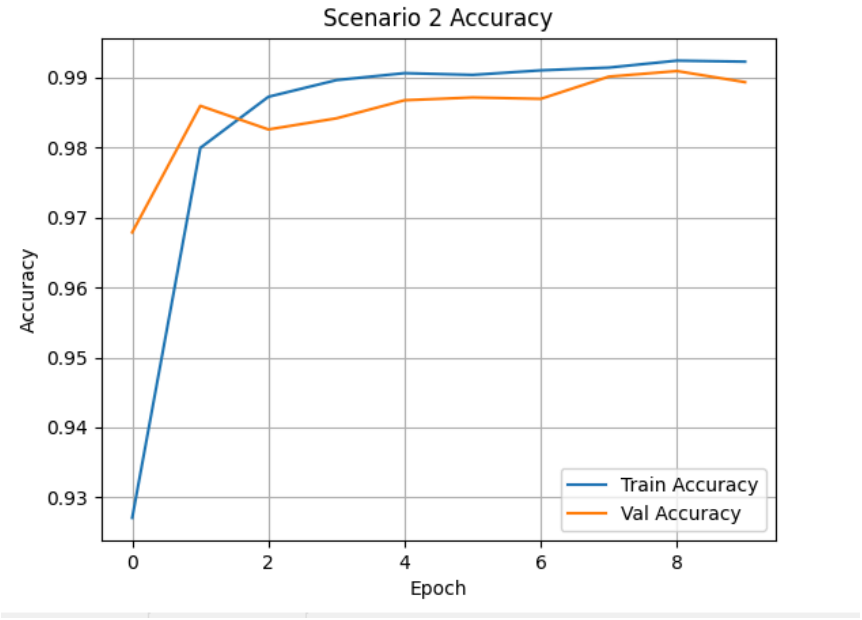
8. Screenshot 8: Scenario 1 Accuracy and Loss Percentages

```
Scenario 1 Test Loss: 0.0272
Scenario 1 Test Accuracy: 99.16%
```

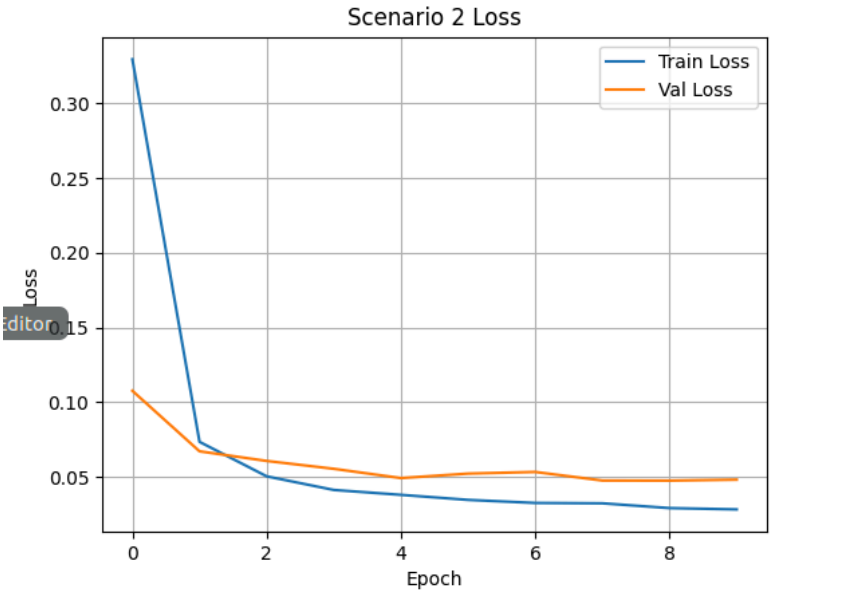
9. Screenshot 9: Running Scenario 2

```
===== Running Scenario 2 =====
[2] X_test shape: (125969, 118)
[2] y_test shape: (125969,)
X_train shape: (25191, 118)
X_test shape: (125969, 118)
Train on 20152 samples, validate on 5039 samples
Epoch 1/10
```

10. Screenshot 10: Scenario 2 Accuracy



11. Screenshot 11: Scenario 2 Loss



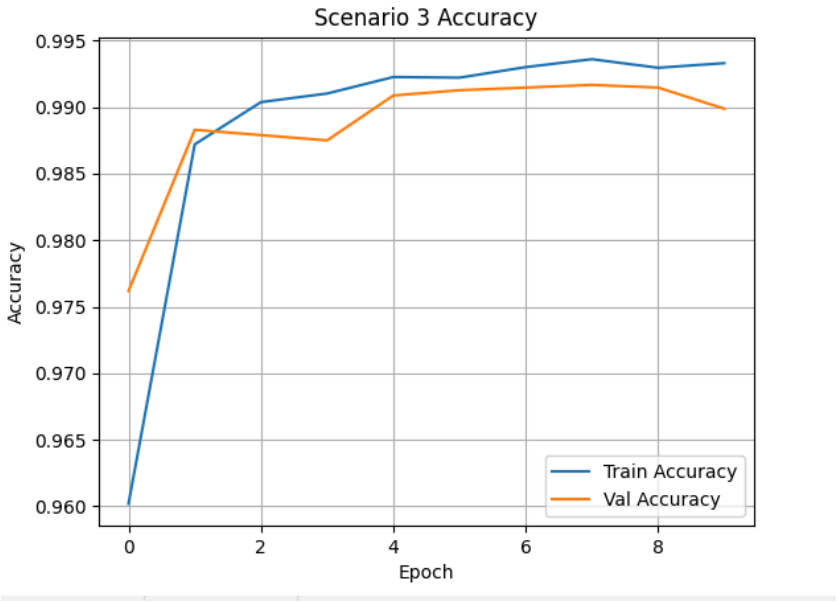
12. Screenshot 12: Scenario 2 Accuracy and Loss Percentages

```
Scenario 2 Test Loss: 0.0391
Scenario 2 Test Accuracy: 99.04%
```

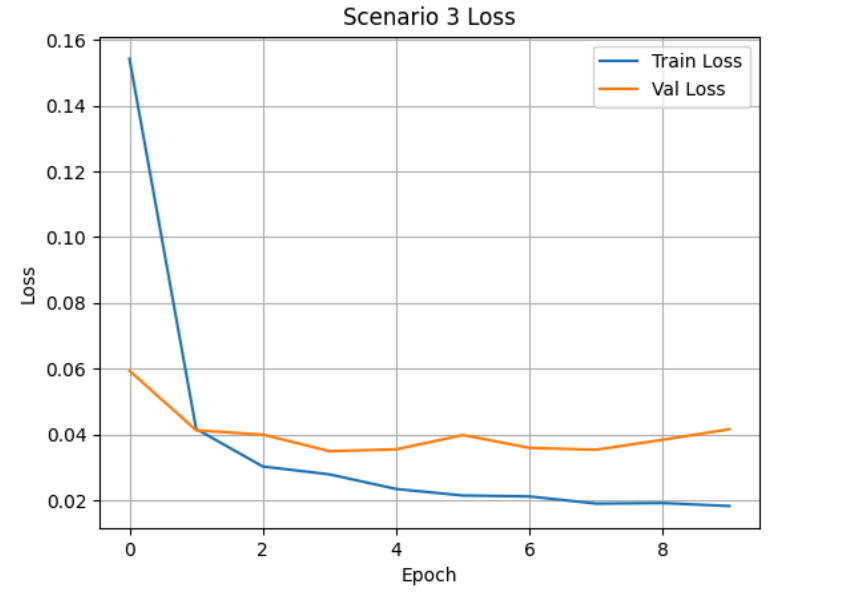
13. Screenshot 13: Running Scenario 3

```
===== Running Scenario 3 =====
[3] X_test shape: (125972, 118)
[3] y_test shape: (125972,)
X_train shape: (25191, 118)
X_test shape: (125972, 118)
Train on 20152 samples, validate on 5039 samples
```

14. Screenshot 14: Scenario 3 Accuracy



15. Screenshot 15: Scenario 3 Loss



16. Screenshot 16: Scenario 3 Accuracy and Loss Percentages

```
Scenario 3 Test Loss: 0.0340
Scenario 3 Test Accuracy: 99.04%
```

17. Screenshot 17: fnn_sample.py (attached to report submission).

VII. References

[1] Python Software Foundation, available at https://www.python.org/, accessed by 06/29/2025.
[2] Sabur Hossain, Github lab files, available at https://github.com/SaburH/CSE548/raw/main/lab-cs-ml-00301.zip, accessed by 06/29/2025.
[3] TensorFlow Open Source Machine Learning Framework, available at https://www.tensorflow.org/, accessed by 06/29/2025.