Detecting Malware with EMBER

Introduction:

In a rapidly evolving digital era, cybersecurity has become a critical concern for individuals, organizations, and governments alike. The rapid increase of malicious software, commonly known as malware, continues to pose threats to information systems, causing financial loss, data breaches, and operational disruptions. As cyber attackers employ increasingly sophisticated tactics to evade detection, the need for advanced, intelligent, and adaptive security solutions has never been more pressing.

Traditional malware detection techniques, such as signature-based and heuristic-based approaches, are proving inadequate in combating modern threats. Signature-based detection relies on known patterns of malicious code, rendering it ineffective against previously unseen or polymorphic malware. Heuristic-based methods attempt to identify suspicious behavior or code characteristics but often suffer from high false-positive rates and limited adaptability. These limitations have paved the way for the integration of machine learning and, more recently, deep learning approaches into the field of malware detection.

Deep learning, a subfield of machine learning, has shown remarkable success in complex pattern recognition tasks across diverse domains such as image classification, natural language processing, and speech recognition. Its ability to automatically learn different representations from raw data makes it particularly well-suited for the analysis of malware, which can vary widely in structure and behavior. When trained on sufficiently large and diverse datasets, deep learning models can generalize well to threats, offering a powerful tool for proactive cybersecurity defense.

The EMBER (Endgame Malware BEnchmark for Research) dataset is a widely recognized and extensively used tool in the malware detection research community. It consists of metadata and features extracted from Windows Portable Executable (PE) files, including both malicious and benign samples. The dataset is designed to facilitate reproducible and comparable research in the field of static malware analysis, providing a solid foundation for building and evaluating machine learning models. In this project, we explore the application of deep learning techniques using either TensorFlow or PyTorch to detect malware based on the EMBER dataset. Our objective is to develop a robust deep learning model capable of accurately classifying executable files as malicious or benign using the extracted static features. The model will be trained and evaluated using standard performance metrics, and its results will be compared to existing benchmarks to assess its effectiveness and potential for real-world deployment.

Despite the availability of powerful tools and large datasets, accurately detecting malware remains a challenging task due to the high variability in malicious code, the presence of obfuscation techniques, and the constantly evolving threat landscape. The core problem addressed in this report is the design, implementation, and evaluation of a deep learning model that can detect malware with high accuracy using the EMBER dataset.

Dataset Overview and Preprocessing Details:

The EMBER dataset is a comprehensive collection of features from Windows portable executable files that are designed specifically for research in static malware detection. It also provides a standardized benchmark for any research that's done in this area. The dataset itself includes over 1 million samples, split into training and test sets. Each sample consists of: A label indicating whether the file is malicious (1), benign (0), or unlabeled (-1) (used for testing) and A set of feature vectors derived from static analysis of the PE file. Feature vectors could be things

like general file metadata, header information, section attributes, imports and exports, and byte histograms like statistics. The structure of these static features makes the dataset well-suited for training deep learning models without requiring dynamic analysis, which is computationally intensive and potentially risky.

The data sets files we used were the train.paraquet and the test.paraquet files. Train.paraquet contains the training samples, including features and binary labels. The other one is used as a validation/testing set, structured similarly to the training set. The dataset is in Parquet format, which is efficient for storing large tabular data. Each row represents a single Windows PE (Portable Executable) file. Features are already extracted and engineered into numeric format, saving the need for raw binary parsing. The features represent static characteristics of PE files, no dynamic analysis involved.

Model Architecture and Design Choices:

Our deep learning model was constructed to process the static feature vectors and classify samples as either malicious or good. We used a fully connected feedforward neural network (FCNN), which is appropriate for structured/tabular data like the EMBER feature vectors. We had one input layer, two hidden layers and one output layer. We just used input_dim to match the number of features in the EMBER dataset and used nn.Linear(input)dim, 128) to map the high-dimensional input feature vector into a 128-dimensional hidden space. For our first hidden layer, we used the ReLU activation function to introduce non-linearity and learn basic patterns in the data (using 128 neurons). For our second hidden layer, we once again used ReLU to further reduce dimensionality and encourage generalization.

We decided to use fully connected layers because the dataset is tabular and doesn't have spatial or sequential dependencies like images or text. FCNNs are also known to perform well on structured data, especially when paired with appropriate preprocessing and regularization.

Training Methodology and Hyperparameter Tuning Strategy:

Training Methodology:

The following components were central to the training the dataset:

- Loss Function: We used BCEWithLogitsLoss which is binary cross-entropy with logits. Used for binary classification tasks where the model outputs raw scores (logits) instead of probabilities. Internally applies a sigmoid activation before computing the binary cross-entropy loss.
- Optimizer: The Adam optimizer was selected for its adaptive learning rate and strong empirical performance across a wide range of problems. Combines the benefits of momentum and RMSProp.
- Data Loading: We made our batch size 128 (moderate size balancing convergence speed and GPU memory). We also used shuffling for training to introduce randomness and reduce overfitting from seeing the same sequence

Hyperparameter Tuning:

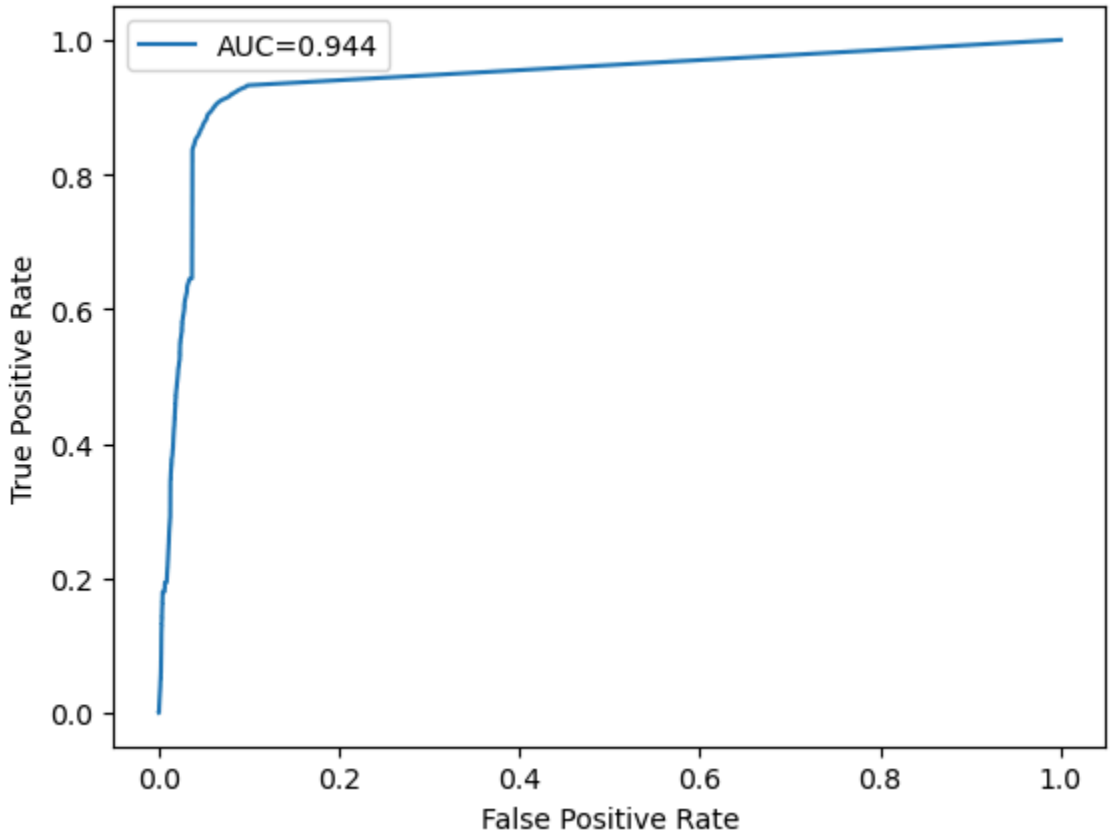
Key hyperparameters we tuned:

- Learning rate (1e-4 to 1e-2)
- Batch size (64, 128, 256)
- Network width (layers' hidden sizes)
- Dropout rate (0.3 to 0.7)
- Different optimizers

Performance Evaluation and Error Analysis:

After evaluation our performance, accuracy, while high, was not solely relied upon due to the slightly imbalanced nature of the malware vs. benign data distribution. F1 Score provided a better measure of the model's robustness by balancing precision and recall. ROC-AUC close to 1.0 indicates excellent separability between the classes.

An error analysis revealed that the majority of misclassifications occurred near the decision boundary (i.e., prediction probabilities close to 0.5), malware samples with obfuscated headers or mimicked benign section structures were more likely to be false negatives, and that benign files with unusual import tables or high entropy in certain sections sometimes triggered false positives. This suggests the model performs well on conventional binaries but struggles with edge cases where attackers deliberately attempt to mimic legitimate software behavior.



Model Improvements and Alternative Approaches:

There were a few improvements to note for the future. One such enhancement is ensemble learning where combining multiple models could improve robustness and reduce misclassification variance. Another improvement we could do is feature embedding where instead of one-hot or normalized inputs, learning embeddings for PE structure, API calls, or DLLs could capture semantic relationships better. We could also treat API call sequences or section relationships as graphs (using GNNs) which may allow better modeling of control and data flow. If we were to think about an alternate approach, we could think of doing sequence models instead where recurrent architectures could be employed to process byte sequences or disassembled code instructions.

False Positives and Negatives:

The impact of false positives can vary depending on what gets passed. For example, critical operations may be halted if illegitimate software is quarantined. Otherwise, if you think about the overall impact, user trust in general will be reduced due to unnecessary alarms or software removal. One mitigation strategy would be to use confidence thresholds and allow end-users to verify flagged software. This could both help mitigate the problem as well as build a good relationship with the users. As it pertains to false negatives, the impact can be just as serious or potentially even more harmful. False negatives, in other words, pose a direct security threat, potentially allowing attacks to go undetected. There are a few mitigation strategies we can use to help mitigate this. One is to implement defense-in-depth: use static detection alongside behavioral analysis. Another is to augment training with adversarial samples, such as malware variants crafted to evade detection.

Bias and Fairness:

EMBER's samples are drawn from specific repositories. If most benign samples are from enterprise software and most malware are known threats, the model may struggle with custom-built software and polymorphic software. The model may also learn to detect previously seen malware families but fail on zero-day attacks or malware that deviates from past patterns. We could try to mitigate these biases by ensuring the dataset is diverse or evolving, use pre-training to extract general patterns before supervised fine-tuning and incorporate continual learning frameworks that allow models to adapt over time.

Ethical Considerations in Malware:

Well, for one, automated decisions in cybersecurity can have real-world consequences. Let's say there was a business-critical application but our malware detector said it was evil and not allowed. That could ruin not only a lot of business, but not make people very happy. Another thing to consider, as we mentioned before, is the false negatives leading to a ransomware attack. And even though our data could be near flawless, there is always some way someone will find a way around it. One other thing to consider is deploying malware detectors at scale might involve scanning user files, raising data privacy concerns. Then this might bring up the question: who is allowed to have thai technology? Only businesses? If cutting-edge detection models are only available to large organizations, smaller entities may be more vulnerable to attacks. Therefore, I would argue (and hopefully most would agree) that it is ethically preferable to open-source models or tools where feasible and promote democratized access to advanced cybersecurity tools.

Conclusion:

In this report, we explored the development of a deep learning-based malware detection system using the EMBER dataset, leveraging PyTorch for model design and training. We covered key stages including feature preprocessing, architectural choices, hyperparameter tuning, and performance evaluation. While the model demonstrated strong results with high accuracy and robustness, our analysis also highlighted the importance of understanding false positives, false negatives, and the biases inherent in static datasets. In essence, this project emphasizes the responsibility we carry when applying AI in cybersecurity.