Deep Learning Project Final Report

Group Members:
Wenbo Wang
Asaf Correa
Michael Conner

Introduction

Malicious PDFs are still a very big way that people fall for malware distribution, phishing, ransomware, and more attacks. A lot of current antivirus software uses what is called signature-based detection. Signature-based detection basically has files, like PDFs, that have a certain digital signature and if that signature is missing or changed, then it will be flagged by the antivirus. The problem with this is that the internet is an ever-changing place. There are new files being created and new obfuscated threats coming out daily. This current system is outdated so something using artificial intelligence and deep learning is a must in order to detect these malicious documents.

To address these problems, our project has a deep learning malware detection system where we can recognize patterns that malicious PDF files have without relying on the outdated file signature system. By training a neural network with raw byte data from benign and malicious PDFs, the model is able to detect patterns or relationships that a human or signature system could never detect. **We used Tensorflow to create a GPU-accelerated deep learning AI model using dense neural networks to efficiently detect malicious PDF files based on raw byte data without relying on traditional signature-based methods.** With this, we achieved around a 98% accuracy in detecting malicious pdf files.

One of the prior works that we reviewed was *Malware Detection by Eating a Whole EXE* released in 2018. This paper proposed the idea of using raw bytes from executable files in order to classify malware. This work basically proved that models can classify files based on raw binaries without having to do preprocessing. We took this information from the paper and basically used the same principle on our PDFs. We used simple dense layers and arrays in order to maximize resource efficiency.

Another paper that we looked at was *Malware Classification using Deep Neural Networks* which looked at the use of DNNs for static malware detection while not requiring the execution of potentially dangerous code. When first start this project I did try opening some of the malicious PDFs but Windows would not allow me. I did not realize that we were dealing with actual malicious files. This article talks about extracting the raw binary data of the file and allowing a DNN to identify malicious patterns without any costly preprocessing steps. The approach in this article used a high-dimensional representation of the files which make it more adaptable to new unseen malware. We saw that DNN could be a very viable option for us because of complex relationships between raw binaries.

The last paper we took inspiration from was *Detection of Malicious PDF Files using Machine Learning Techniques* which took a more manual approach even though it was still machine learning. This approach extracted structural features like number of objects or streams and classified those into a machine learning algorithm like random forests. We decided to just go all in with our deep learning model and let the network determine what bytes were malicious or not.

Background

This project implements a supervised deep learning system using a Dense Neural Network **(DNN)** to classify binary files such as EXE and PDF as either benign or malicious. In order to understand the implementation, we need to learn some key principles in deep learning, binary data handling, neural network architecture, and model evaluation with the help of modern machine learning libraries such as TensorFlow and Keras

In traditional machine learning tasks, the input data is typically structured, such as CSV tables or image matrices. By comparison, this project deals with unstructured binary data. Each file is read as a stream of raw bytes and transformed into a fixed-length numerical vector using NumPy. Since binary files vary in length, padding or truncation is applied to create uniform inputs of 152000 bytes per file. Each byte is scaled to the **[0, 1] range** to help with numerical stability and make sure the model gets consistent input. This step is key for turning raw binary into format neural networks can learn from.

The architecture of the DNN consists of a sequence of fully connected 'Dense' layers. The first hidden layer contains 512 units with **ReLU activation**, followed by a second hidden layer of 128 units, also using ReLU. The final output layer consists of a single neuron with sigmoid activation function, which outputs a probability indicating the likelihood that the input file is malicious.

Model training is executed using the **'model.fit' function** in Keras. The model is compiled with the **Adam optimizer**, which adapts the learning rate for each parameter and is known for fast convergence. The binary cross-entropy loss function is used, which measures the error between predicted probabilities and the true binary labels. The training process is related to iterating over batches of data for multiple epochs, adjusting weights to minimize the loss function.

Following training, the model is evaluated on a separate test dataset using 'model.evaluate', which offers metrics such as test accuracy and loss. In order to visualize the model's performance over time, we need to use Matplotlib to plot the accuracy and loss for both the training and validation sets across epochs. These plots help diagnose issues such as overfitting or underfitting and guide further model tuning.

The whole system is built using three core libraries: TensorFlow/Keras for model construction and training, NumPy for efficient binary data processing and manipulation, and Matplotlib for performance visualization. This combination of tools allows for streamlined implementation, scalability, and ease of experimentation.

Furthermore, it is important to recognize the design trade-offs inherent in this approach. Dense Neural Networks, while simpler and computationally less expensive, are limited in their capacity to exploit local or hierarchical patterns in input data. In more advanced architectures like CNNs, spatial relationships between nearby features are preserved through the use of kernels and receptive fields. In contrast, our model treats each byte in

the binary sequence as an independent feature, which may overlook structural relationships in the data such as repeating patterns, headers or embedded scripts.

In conclusion, the project integrates data standardization, neural network modeling, supervised learning, and diagnostic visualization into a complete malware detection pipeline. Despite its simplicity compared to more advanced architectures like Convolutional Neural Networks, the Dense Neural Network provides a practical, interpretable, and computationally efficient solution for detecting malicious patterns in raw binary data.

Data and Model

The goal of this project was to build a binary classifier capable of distinguishing benign PDF files from malicious ones with high accuracy and a low rate of false alarms. The dataset contained 99 145 PDF files each no larger than 150 KB and split into 55 450 training samples, 4 305 validation samples and 25 783 test samples. Each file carried a label of zero for benign or one for malicious.

Each PDF was read as a raw byte sequence and resized to a fixed length of 152 000 bytes by padding with zeros or truncating longer files. The byte values were scaled to the range zero to one by dividing by 255 and cast to 32-bit floats. This simple normalization prepared the data for direct input into the network.

The model was a feed-forward neural network with three fully connected layers. The first layer had 512 units with a ReLU activation, the second layer had 128 units with a ReLU activation, and the output layer produced a single probability through a sigmoid activation. The network had 77 890 305 trainable parameters, reflecting the high dimensionality of the raw-byte input.

Training ran for ten epochs using the Adam optimizer, binary cross entropy loss and a batch size of 32. After the first epoch the training accuracy was 96.96 percent and the validation accuracy was 96.79 percent. By the third epoch training accuracy had risen to 98.39 percent and validation accuracy reached 98.19 percent. During the fifth epoch validation accuracy dipped to 93.98 percent before recovering in subsequent epochs. Final validation accuracy at epoch ten was 97.70 percent.

When evaluated on the held-out test set the model achieved a loss of 0.0827 and an accuracy of 97.75 percent. These results demonstrate that the network learned to identify malicious PDFs with a level of performance consistent with the project aim. While overall accuracy is high, further analysis of true positive and false positive rates can refine threshold selection for different operational requirements.

Experiments and Analysis of Results

There were a lot of small technical holes that we had to jump through in order to get everything working. I think the easiest part that ended up taking up a lot of time was getting the GPU (graphics card) to work with Tensorflow on Jupyter Notebook. You would think something like this would just work right out of the box

but there are actually a lot of steps you have to take and make sure a lot of versions are correct. For one, the newest versions of TensorFlow on Windows do NOT support GPU capabilities unless you are using a VM that has Linux on it. This information was not very wide spread so this was the quickest fix that took a long time to come to. We used TensorFlow 2.10 and from there we had to use an older version of CUDA and cuDNN from Nvidia. The versions of TensorFlow have to match up with these very precisely. cuDNN also has specific download links for older versions that are behind a developer portal just making it more complicated. Once all of those are correct you need to uninstall tensorflow and numpy because likely your tensorflow compiled with Numpy 2.x.x, which is incompatible with tensorflow 2.10 it uses a 1.x.x version. So reinstall both, and then things should work. Very complicated but the effort is worth it, running everything 30x faster than a CPU would because of the VRam and a GPU's multi-tasking capabilities.

We used raw byte level data from PDF files and either padded or truncated (mostly all padded) all of the files to 152,000 bytes. Then we organized the data into train test splits. We trained the network on around 64% of the data, tested on 30% and validated on 5% which was around 100,000 files total. A small batch size was used because the GPU kept crashing the kernel when it would run out of VRam. We found the accuracy of the model dropped when we decreased the size of Bytes we take in so we reduced the batch size. We decided to go with 10 epochs because we did not want to overfit the data but still wanted it to have a good overview of everything. The GPU also worked much much faster after the first through third epochs because of TensorFlow's caching capabilities.

After all of this we ended up with a model getting around 98% accuracy on predicting malicious files from benevolent ones.

We think this high level of performance can be attributed to a few things. We were able to learn from our research what the best technique would be. Raw byte level patterns probably preserved low-level structures that probably would get lost or dismissed in feature engineering. Our data set given to us by the professor was also vast enough that there was a lot of time for the network to normalize itself, even though it still stumbled around epoch 5. We think the dip around epoch five was from temporary overfitting that the model recovered from very quickly.

From taking a lot of time to get the GPU working to selecting which model we would use to working on the actual model itself we are very proud with the accuracy that we were able to attain. We are very curious to see how other groups did and what accuracy percent they were able to achieve and through what models they used.

Roles

Michael Conner: My role was focused a lot on fine-tuning the model, getting the GPU to work with Tensorflow, and a lot of bug fixing. Choosing things like the batch size, how much data to read, and the epoch count, and trying to configure everything to work smoothly. We all worked on the code, but I was the main one to add the GPU acceleration so that we would be able to train our model faster. There were also a few problems with getting the files to read properly. Every few batches, there would be a file that did not read, and we ended up making the call to drop those files

because we thought our model would still be strong without them. That obviously came with the challenges stated above. We all individually worked on parts of this paper and our presentation.

Wenbo Wang:
I take the initiative to reach out to all team members in an effort to coordinate and move our project forward. Given that everyone has a busy and often conflicting schedule, it has been quite challenging to find a time that works well for everyone. Despite these difficulties, I actively propose several solutions to help improve our project. At the same time, I take the time to analyze and compare the different project ideas contributed by each team member. My goal is to identify the most effective combination of elements from each proposal, recognizing that every project has its own strengths and weaknesses. By doing this, we aim to foster collaboration and make sure we develop a well-balanced and high-quality final outcome that reflects everyone's contributions.

Asaf Correa:
Throughout the project, I provided focused recommendations on our shared codebase and ran my own experiments to validate each change. I developed test scripts that exercised different implementations, gathered results on stability and performance, and used that evidence to refine my suggestions. Once I had concrete data, I reviewed every teammate's code, identified the most effective solutions from their contributions, and merged those into a single, unified model. This careful process of experimentation, feedback, and consolidation ensured our final implementation combined the best ideas from the entire team.

Sources

Raff, E., Barker, J., Sylvester, J., Brandon, R., Catanzaro, B., & Nicholas, C. (2018). Malware detection by eating a whole EXE. arXiv preprint arXiv:1710.09435. https://arxiv.org/abs/1710.09435

Smith, J., Chen, L., & Patel, R. (2023). *Malware Classification using Deep Neural Networks.* arXiv preprint arXiv:2310.06841. https://arxiv.org/abs/2310.06841

Li, W., Zhong, W., & Zou, D. (2017). Detection of malicious PDF files using machine learning techniques. 2017 2nd International Conference on Anti-Cyber Crimes (ICACC), 185–190. https://doi.org/10.1109/ICACC.2017.794268