Exploring Optimizations of the EMBER Dataset Baseline Model

Henning Sara Strandå *henninss@stud.ntnu.no*
Norwegian University of Science and Technology,
Department of Computer Science,
NO-7491 Trondheim, Norway

Abstract

The EMBER dataset is a labeled dataset consisting of information generated from malicious, benign, and unlabeled Windows executables, which is meant to function as a benchmark for malware-detecting machine learning models.

This paper describes brief forages into other machine learning architectures applied to the dataset, but mainly optimization of the gradient-boosted decision tree baseline model. This includes looking at previous work done using deep learning methods, and documenting the techniques used (mainly hyper-parameter optimization and feature selection) for achieving better performance and accuracy for the benchmark model.

We find that marginally better accuracy than the baseline model is available with advanced and significantly more computationally intensive models based on neural networks, making optimizing these models by method of grid-search exponentially difficult. Optimizing a gradient-boosted decision tree, which has a significantly lower performance footprint, we very quickly approach a better accuracy than most deep learning methods. The benefit of a GBDT is that feature selection is not as crucial for model performance than in neural networks.

The findings imply that gradient-boosted decision tree models are significantly more performance-efficient for the task of detecting malware, and also easier to optimize. It remains somewhat unclear whether better performance is still achieved in more computationally intensive models, but with a significantly higher training time.

The source code for the methods used in this paper can be found here.[1]

1 Introduction

1.1 Dataset

In an increasingly digital world, detecting malware is essential to facilitate the safe use of computers, both for professional and personal use. Methods for detecting such malware are therefore of great interest.

The EMBER dataset (Anderson & Roth, 2018) is one of the most popular labeled datasets for malware, and was released with the intention of being a benchmark made specifically for the current machine learning landscape. The dataset contains data extracted from over 1 million Windows binaries in the PE (Portable Executable) format, which are labeled as either benign, malicious, or unlabeled.

The data extracted from binaries are of very different types. A number of features are summarized by hashing. An example would be the exported functions in a binary; This feature is summarized as a list of function names, which are then hashed. This makes it easier to vectorize and sequence, the details of which are explained in section 1.3. The actual bytes of predetermined sections of the underlying executable are summarized by way of a byte histogram, as well as byte entropy histograms.

[1]Source code: `https://github.com/emasru/ml-project-ember-optimizations`

1.2 Evaluation of models

ROC AUC (Receiver Operating Characteristic Area Under the Curve) is a performance metric for binary classification models. The ROC is represented as a graph that plots the True Positive Rate (TPR) against the False Positive Rate (FPR) at various threshold levels, and is a more accurate metric for how well a model distinguishes between classes. Below are formulas for calculating the TPR and FPR, respectively:

$$\text{TPR} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

$$\text{FPR} = \frac{\text{False Positives}}{\text{False Positives} + \text{True Negatives}}$$

The AUC score is obtained from the area constructed by the ROC graph, and is particularly useful when we are interested in minimizing the FPR in our model. Higher is better. The AUC score will be used as an optimizing function in our baseline-derived models.

To evaluate our models, we will be using the TPR and FPR for both classes in the test set (malicious vs benign)[2], as to not have to evaluate the model at various thresholds, and instead is only calculated at the maximum AUC score (since this is the metric being maximized in the model). The TPR and FPR for both classes form the "confusion matrix" of the model.

1.3 Baseline model

Along with the original paper, Anderson & Roth released source code[3] for constructing a baseline model by vectorizing the dataset, as well as helper functions for optimizing the model and extracting features. The vectorization of the different types of features present in the dataset results in a vector consisting of 2351 dimensions, where the authors assuredly note that a significant number of the features are too noisy or irrelevant for the problem domain altogether. The feature names and what they represent are described in-full in the original paper.

The baseline model is a gradient boosted decision tree model (GBDT), using the LightGBM framework[4].

Label	TPR	FPR
Benign	92%	5%
Malicious	95%	8%

Table 1: Confusion Matrix for benchmark model with all features

GBDT's are notable in that they are a much older and simpler type of model compared to deep learning methods.

1.4 Objective

Anderson & Roth notes that:

> "Simple approaches to immediately improve model performance include feature selection to eliminate noisy features and hyper-parameter optimization via grid search."

The objective of this paper is to explore methods for either lessening the computational power needed to achieve the same model accuracy, or achieving a higher model accuracy.

[2]The training set contains unlabeled binaries, but considering they are not present in the testing set, they are disregarded for the purposes of this study

[3]The source code for the EMBER baseline model (as well as the dataset) can be found at: `https://github.com/elastic/ember`

[4]Documentation for the LightGBM framework: `https://lightgbm.readthedocs.io/en/stable/`

2 Related Work

2.1 Gradient Boosted Decision Trees vs. Neural Networks

Although GBDT's have classically been well-suited for "tabular" data problems, Neural Networks (NN) and deep learning methods have increasingly been applied to these problems. It is still somewhat unclear which method class is better for the problem type, the problem of which is thoroughly examined in McElfresh et al. (2024). Interestingly, the paper notes that fine-tuning of hyper-parameters in a GBDT usually yields the same gains in model accuracy as switching to a NN, or a negligible difference.

2.2 Neural network optimizations

However, it is still of interest to examine the results of using NNs on the EMBER dataset. One such attempt (ALGorain & Alnaeem, 2023) used deep learning algorithms, and optimized the model using grid-search and covering arrays. One grid-search explored in this paper had ranges of 1200-2400 neurons over 3 layers, and learning rates from 0.001-0.2, among other hyper-parameters. The optimal configuration found through cAgen had three layers consisting of 2400, 1200, and 1200 layers respectively, with a learning rate of 0.1. The search was conducted on (either) a iMac (Retina 5K, 27-inch, 2017, 4.2 GHz Quad-Core Intel Core i7, 16 GB 2400 MHz DDR4 RAM, and a Radeon Pro 580 8 GB graphics card) and Windows OS 11, with 11 Gen Intel Core i7-11800H 2.30 GHz processor, with 16 GB RAM.

The search was reported to take 29 days, with a model accuracy of 95.42%. The paper does not mention details on the FPR.

3 Methods

All methods in this sections are implemented using Python. For graphing and various utilities, Matplotlib and Scikit was utilized. The source code can be found at [5]. The `README.md` for the repository contains and overview of the contents of each file as it relates to this section, and also installation instructions for setting up the environment using Conda.

3.1 Hardware specifications

All methods listed in this section was conducted on Manjaro Linux (based on Arch Linux) with a Ryzen 7800x3d CPU, with 32GB of RAM. Due to constraints with the EMBER dataset source code and differing CUDA versions, hardware acceleration was not used.

3.2 Grid-search

"Grid-searching" consists of systematically evaluating a set of values for multiple hyper-parameters by testing every combination of the different value sets. A selected set of values for the hyper-parameters are then evaluated, and the result stored, so at the end they are able to be ranked. We will look at only doing a grid-search for the baseline model, see section 3.3.

Although it is not mentioned in the paper [6], the source-code for the original paper contains a library function `optimize_model` for optimizing the hyper-parameters[7] used in the benchmark model.

[5] See footnote 1.

[6] The commit that added a helper function for optimizing the benchmark model can be found at: `https://github.com/elastic/ember/commit/7cbbdac7674b0ffb57963b95283d15e6b5fb464e`. Note that this commit is approximately 1 year older than the original paper.

[7] See footnote 3.

Parameter Name	Values Tested
num_iterations	500, 1000
learning_rate	0.005, 0.05
num_leaves	512, 1024, 2048
feature_fraction	0.5, 0.8, 1.0
bagging_fraction	0.5, 0.8, 1.0

Table 2: Values searched for the gradient-boosted decision tree model

The `num_leaves` value is the number of leaves that the decision tree is able to grow to. `feature_fraction` represents a fraction of the features that are present in training the model, and a lower fraction will randomly select features to be used to train the model that is being searched. This is random for every search iteration, and ensures that over the course of the whole grid-search, we can reasonably conclude whether or not a model on average benefits from fewer features without actually doing any work on selecting features yet.

Also of interest, the `bagging_fraction` is used to introduce "bagging" into training. Bagging is when data points are able to be presented in training more than once (drawing from the dataset with replacement), and in general has the potential to speed up training and even prevent over-fitting.

3.3 A brief look at neural networks

Given the time-frames presented by ALGorain & Alnaeem, systematic optimizations of NNs are considered out of scope for this paper, and will therefore not be explored at length. We will however, briefly look at constructing a Recurrent Neural Network (RNN) using TensorFlow with only 128 neurons in the first layer. The choice of an RNN for this task is mainly based on an assumption that such a model will be able to identify sequences in the dataset, which in a binary can be repeating sections of machine code. Even though sections are divided and hashed instead of actually represented in the dataset, a large enough repeating section should be able to trigger such a pattern.

The training of the RNN is set to abort if model accuracy deteriorates over 2 epochs.

3.4 Feature selection

For our RNN, we use Select K-best (SKB) to find the top features in the vectorized dataset, using the ANOVA-F value to determine statistical significance.

In the baseline decision tree model, we instead train the model using all features initially. Afterwards, we use LightGBM's `feature_importance` method for extracting the features that result in the most gain in AUC score for our model. These selected features are now used to train a new decision tree.

The "gain" in a decision tree measures how much a feature reduces the loss function when it is used to split the data. This is straightforward to compute in decision trees, as the splits are discrete, making it easy to quantify the reduction in the loss function at each node.

4 Results and Discussion

4.1 Recurrent Neural Network

Even after tweaking and experimenting with different hyper-parameters, training on the RNN would consistently lead to early over-fitting of the model. We observe that even with a learning rate on the lower end of what was found by ALGorain & Alnaeem, our model which in this case only consists of 128 neurons in the first layer, is already overfitting. After only 4 epochs of training with a learning rate of 0.001, we reach a consolidated TPR of 86.86% and a FPR of 20.53%. Adding more layers signficantly increase the time to train the model, but would still lead to early overfitting.

4.2 Grid-search on benchmark model

Parameter Name	Value
num_iterations	500
learning_rate	0.05
num_leaves	1024
feature_fraction	0.5
bagging_fraction	0.5

Table 3: Best performing model found through grid-search for benchmark model

Label	TPR	FPR
Benign	98%	3%
Malicious	97%	2%

Table 4: Confusion Matrix for most optimal benchmark model with all features (features dropped randomly)

Comparing the results from the benchmark model (Table 1) with the optimal model, we see a considerable improvement to both the TPR and FPR. It is interesting to note that we kept the `feature_fraction` parameter described in section 3.2, which randomly discards half the features, and we are still able to get improved results.

A total of 321 models were evaluated in the grid-search. Due to the thorough cross-validation implemented[8], evaluating each model took longer than the normal time it would take to train, leading to the whole search taking a little over 48 hours on the setup described in section 3.1. Of interest is also the average score[9] for each value of the hyper-parameters across all iterations:

Parameter Name/Value Pair	Average Score
learning_rate = 0.005	0.549
learning_rate = 0.05	0.568
num_iterations = 500	0.556
num_iterations = 1000	0.561
feature_frac = 1	0.554
feature_frac = 0.8	0.558
feature_frac = 0.5	0.563
num_leaves = 512	0.556
num_leaves = 1024	0.559
num_leaves = 2048	0.560

Table 5: Average Score for each hyper-parameter value across all iterations

4.3 Feature selection

4.3.1 Select K-best

As the features selected for the RNN did not yield satisfactory results, the features that were selected by SKB are not particularly significant. However, it is still interesting to note which features were considered the most statistically significant. A list of the 100 most significant features can be found in the source-code. The highest ranking features appear to be sections of the byte histogram (implying there are certain frequencies of specific byte-values that signal a certain class of programs).

[8]Training and testing data split using Scikit `TimeSeriesSplit`

[9]This score is supposed to represent the AUC score for the model, but due to erroneously not removing unlabeled data for the dataset they are offset. In the end, this does not matter as much, as a higher score is better

4.3.2 Decision tree feature selection

As described in section 3.4, the `feature_importance` method is used to obtain the following features:

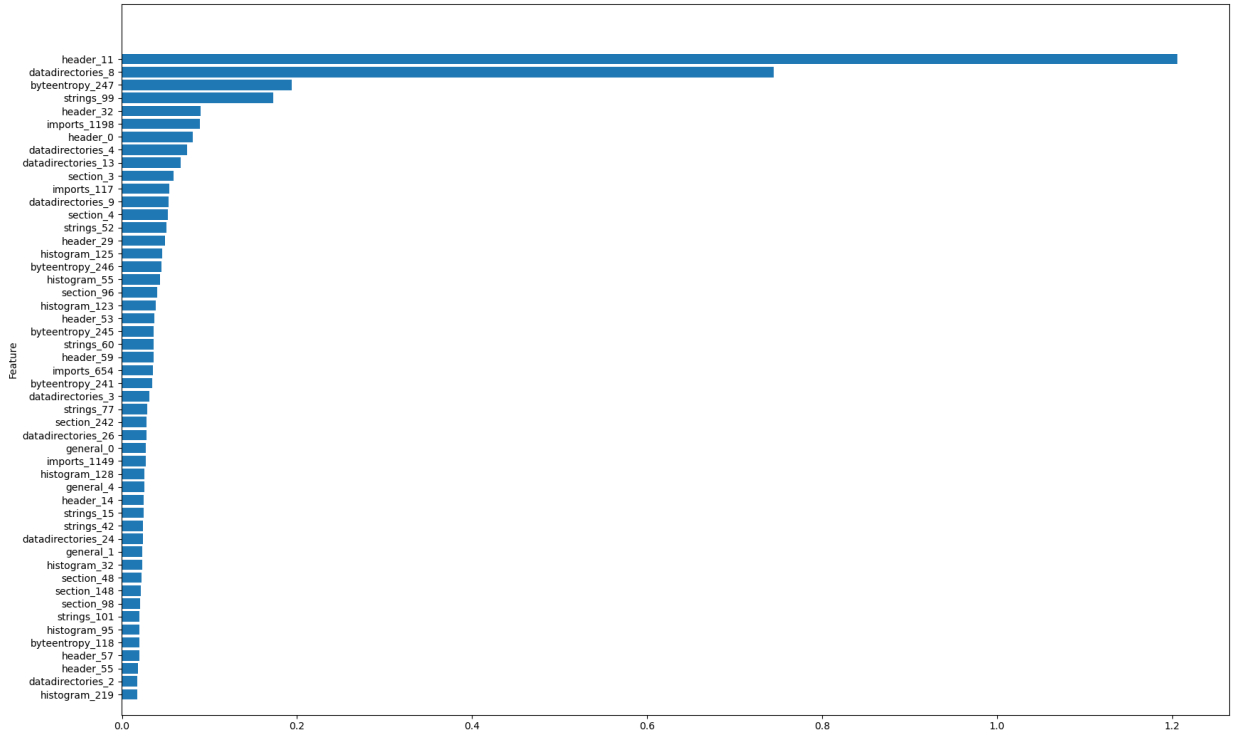

Figure 1: Features which gives the best loss reductions in the GBDT

The set of features that lead to a better model, seem to be very diverse, although it is easy to observe that a certain part of the header (header_13) for the binary and certain pointers to sections (datadirectories_8), are responsible for the vast majority of model performance.

4.4 Performance

Across the board, we observed a big increase in performance when training LightGBM models, than using deep learning methods through TensorFlow. It should of course be noted that with hardware acceleration, the performance cost associated with TensorFlow is lessened considerably.

5 Conclusion and Future Work

Of particular interest to us is the performance gain offered by GBDT's. The offered flexibility of testing different hyper-parameters on-the-fly made experimenting effortless. We also observed that a GBDT with all 2351 features did not take significantly longer to train than a model that had been thoroughly trimmed of redundant features, which is a significant advantage in regards to the usual state of affairs in deep learning. That is all to say, that our findings are consistent with McElfresh et al.'s conclusion that the accuracy gains offered by NN's and other deep learning methods are marginal at best, and detrimental to performance at worst. Tweaking of the hyper-parameters of the baseline model by grid-search remains the most consistent and easy way to improve the model, and for the time being remains the most accessible method for training a model on the EMBER dataset.

Its also interesting to note that the features selected for the neural network and the features for the GBDT differs significantly. One reason could be that decision trees are known to scale better with non-linear feature relationships.

Although the grid-search lead to a significant gain in performance, it still leaves some to be desired. Further work can be done to test different hyper-parameters with selected features, as opposed to random dropouts of features. Considering the original grid-search took 48 hours, such a venture requires better hardware (and/or hardware acceleration), as well as more time. A more thorough examination of deep learning methods also hinge on this.

Since the latest iteration from the dataset is starting to become 6 years old as of the time of publication, it would be interesting to see malware that is not a part of the dataset be tested against the models outlined in this paper. Methods for vectorizing custom binaries remain a part of the EMBER source-code.

References

Fahad T. ALGorain and Abdulrahman S. Alnaeem. Deep learning optimisation of static malware detection with grid search and covering arrays. *Telecom*, 4(2):249–264, 2023. ISSN 2673-4001. doi: 10.3390/telecom4020015. URL https://www.mdpi.com/2673-4001/4/2/15.

Hyrum S. Anderson and Phil Roth. EMBER: an open dataset for training static PE malware machine learning models. *CoRR*, abs/1804.04637, 2018. URL http://arxiv.org/abs/1804.04637.

Duncan McElfresh, Sujay Khandagale, Jonathan Valverde, Vishak Prasad C, Benjamin Feuer, Chinmay Hegde, Ganesh Ramakrishnan, Micah Goldblum, and Colin White. When do neural nets outperform boosted trees on tabular data?, 2024. URL https://arxiv.org/abs/2305.02997.