Computers & Security 137 (2024) 103582

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

Automated machine learning for deep learning based malware detection

Austin Brown *, Maanak Gupta, Mahmoud Abdelsalam

ARTICLE INFO

Keywords:
Malware detection
Automated machine learning
Deep learning
Cloud security
Static malware analysis
Online malware analysis

ABSTRACT

Deep learning (DL) has proven to be effective in detecting sophisticated malware that is constantly evolving. Even though deep learning has alleviated the feature engineering problem, finding the most optimal DL model's architecture and set of hyper-parameters, remains a challenge that requires domain expertise. In addition, many of the proposed state-of-the-art models are very complex and may not be the best fit for different datasets. A promising approach, known as Automated Machine Learning (AutoML), can reduce the domain expertise required to develop custom DL models by automating the ML pipeline key components, namely hyperparameter optimization and neural architecture search (NAS). AutoML reduces the amount of human trial-and-error involved in designing DL models, and in more recent implementations can find new model architectures with relatively low computational overhead.

Research on the feasibility of using AutoML for malware detection is very limited. This work provides a comprehensive analysis and insights on using AutoML for both static and online malware detection. For static, our analysis is performed on two widely used malware datasets: SOREL-20M to demonstrate efficacy on large datasets; and EMBER-2018, a smaller dataset specifically curated to hinder the performance of machine learning models. In addition, we show the effects of tuning the NAS process parameters on finding a more optimal malware detection model on these static analysis datasets. Further, we also demonstrate that AutoML is performant in online malware detection scenarios using Convolutional Neural Networks (CNNs) for cloud IaaS. We compare an AutoML technique to six existing state-of-the-art CNNs using a newly generated online malware dataset with and without other applications running in the background during malware execution. We show that the AutoML technique is more performant than the state-of-the-art CNNs with little overhead in finding the architecture. In general, our experimental results show that the performance of AutoML based static and online malware detection models are on par or even better than state-of-the-art models or hand-designed models presented in literature.

1. Introduction

1.1. Overview and motivation

Malware is becoming a more profitable domain for malicious actors with the rise of digital connectivity and the growing critical infrastructures reliance. These cyberattacks have costed the industry billions (Anderson et al., 2019) of dollars. The increase and impact of such cyberattacks has called for novel and sophisticated defense mechanisms in response to those that wish to protect digital assets from malware attacks.

There are several existing approaches for malware analysis, including static (Nath and Mehtre, 2014; Shalaginov et al., 2018), dynamic (Alotaibi, 2019; Tobiyama et al., 2016; Willems et al., 2007), and online analysis (Kimmel et al., 2021; Kimmell et al., 2021; McDole et al., 2020, 2021). Each of these analysis methods collect different features from the file or system in question, ranging from details of the file header in static analysis, to holistic operating system level performance metrics in the case of online analysis. The reasons to use a specific analysis approach depend on the use case and availability of data. For simple file scanning, static analysis is the fastest method, since there is no need to run the executable in question, whereas, collecting data from a running executable in dynamic analysis may give more insight into the true behavior and intent of a questionable executable. On the other hand, unlike dynamic and online analysis, static analysis can be crippled using well-known obfuscation techniques.

Machine learning (ML), especially DL, has become a popular technique to develop malware detection solutions, and has shown promising results (Sahin and Bahtiyar, 2020) because of its ability to learn generalized patterns to identify unseen malware. As such, research works (Aryal et al., 2021, 2022; Brown et al., 2022; Gupta et al., 2023; Kim-

* Corresponding author.
E-mail addresses: ambrown2000@gmail.com (A. Brown), mgupta@tntech.edu (M. Gupta).

https://doi.org/10.1016/j.cose.2023.103582
Received 7 August 2023; Received in revised form 7 October 2023; Accepted 3 November 2023
Available online 10 November 2023

mell et al., 2021; Kolosnjaji et al., 2016; Mishra et al., 2019; Pascanu et al., 2015; Tobiyama et al., 2016; Xiao et al., 2019; Xie et al., 2020) have employed different types of ML models to detect malware on a variety of systems and data sources, depending on the use case. These proposed solutions have utilized both traditional ML algorithms and, more recently, deep learning algorithms. Approaches (Abou-Assaleh et al., 2004; Fan et al., 2016) that rely on traditional machine learning models require domain experts for feature engineering, which, in most cases, is burdensome and laborious. On the contrary, deep learning based approaches (Agrawal et al., 2019; Ganesh et al., 2017; Jha et al., 2020; Kimmel et al., 2021; Luckett et al., 2016; Raff et al., 2018a; Rudd et al., 2019; Sewak et al., 2018; Vinayakumar et al., 2019; Wang et al., 2019; Yeo et al., 2018) eliminate the feature engineering step and are gaining more traction. Some works (McDole et al., 2020; Rezende et al., 2018) have utilized state-of-the-art DL models (e.g., ResNet (He et al., 2016), DenseNet (Iandola et al., 2014), and VGG16 (Simonyan and Zisserman, 2014)) that perform well in general and train it on malware data; however, these models are usually very complex and require a rigorous tuning process to achieve the desired high performance. In addition, such models are usually designed for tasks like image, text, or voice recognition and can be inadequate for malware detection. Consequentially, works (Raff et al., 2018a) have focused on manually crafting model architectures that fit the malware detection domain. However, these approaches not only require heavy tuning, but also high technical skills in both the ML and the malware domains.

Automated Machine Learning (AutoML) (He et al., 2021) seeks to automate the process of finding an optimal model architecture for the given data and tuning this model to achieve higher performance. In addition, it can also reduce the work needed to redesign a malware detection model as malware and data sources evolve overtime. Even though AutoML pipelines require more computational time to produce a model, they significantly reduce the work hours and expertise needed to find a performant model.

AutoML holds significant promise for malware detection, automating the dual tasks of discovering the ideal machine learning architecture tailored for this purpose and subsequently refining this selected model. Despite its potential, AutoML is still in its developmental phase, and comprehensive exploration of its applicability, especially in malware detection, remains limited.

Building on this, AutoML serves as a potent tool for domain experts, even those without the formal “Data Scientist” designation, enabling them to harness machine learning more effectively. The efficacy of AutoML, like many machine learning realms, often hinges on the volume of accessible data and the intricacy of the given task. Domains rich in data, such as our focus on malware detection, as well as simulated environments that can generate vast amounts of data, are well-poised to reap the benefits of AutoML. While our discussion centers on its utility in static and online malware detection, AutoML’s potential can be extrapolated to other cybersecurity areas like Network Intrusion Detection, Security Log Analysis, and Threat Intelligence. Essentially, any domain with ample high-quality data suited for deep learning might find AutoML advantageous.

With the growth in malware sophistication and machine learning complexity, especially in deep learning, finding the most performant deep neural architecture without a significant increase in human hours spent is critical. This paper aims to study the feasibility of integrating AutoML into the malware detection pipeline to remove the need to hand design and tune ML models. In particular, we focus on using deep learning, specifically Feed Forward Neural Networks (FFNNs) and Convolutional Neural Networks (CNNs). FFNNs have a high level of expressive power and require much less feature engineering than traditional machine learning approaches. Convolutional Neural Networks can model complex functions with image shaped data as input with little to no feature engineering, only requiring framing the data in a 2d vector. Further, we focus on both data that is gathered through static analysis, specifically focusing on portable executable (PE) files which are the predominant executable format in the Windows operating system, and online data captured from running, internet connected, Linux servers in cloud IaaS with malware executed on them. The *main contributions* of this work are:

- We study the feasibility of using AutoML for deep learning based static malware detection and demonstrate the effectiveness of the produced AutoML Deep FFNN models by showing that they are comparable to manually crafted models, even without significantly tuning the AutoML pipeline.
- We provide insights and analysis of the automation parameters of the AutoML process on static malware data, and show how these parameters can affect the performance of the found optimal model.
- We show that AutoML derived Convolutional Neural Networks can preform better than state-of-the-art Convolutional Neural Networks on online malware data, with little overhead in deriving the model architecture.
- We discuss ideas and future directions for improving the efficiency and performance of AutoML models that are designed for malware detection.

The rest of this work is organized as follows. Section 2 discusses background and related works in this domain. Section 3 shows the application of AutoML in two popular static malware datasets, with comparison to other works, and discussion of the presented AutoML methodology. Section 4 focuses on one-shot AutoML applied to CNNs to detect malware in online cloud IaaS, with comparison to detection results of state-of-the-art CNNs on the same dataset. Section 5 presents ideas for future work and improvements, as well as the conclusion to the findings in this work.

2. Background and related works

2.1. Malware detection

2.1.1. Static analysis

Static analysis involves analyzing features that can be observed in a binary without running the binary executable. Static analysis methods may include observations such as: file entropy; n-gram analysis of byte sequences in a binary; imports and API calls; strings found within the binary; header information. The major benefit of static analysis is its speed and low overhead, since the binary is not executed.

One of the most simple forms of static analysis for malware detection is looking up the signature of a binary, most often the file hash. This method is extremely efficient if the binary’s hash is documented, but has no ability to detect modified or new malware. A more popular method of static analysis looks at n-grams of bytes in the binary. Authors in Abou-Assaleh et al. (2004) measured frequency of common n-gram bytes in Windows binaries to determine if the binary is malicious. The frequency of n-grams across both malicious and benign binaries were used to train a K-nearest-neighbors classifier. While this approach showed good results (at the time published), it is unclear if it would show as good of results in modern malware detection. This approach additionally has proven to be computationally expensive and offers diminishing returns as n increases (Raff et al., 2018b). Another work (Fan et al., 2016) has taken it a step further from n-gram byte analysis to analyzing instruction sequences in questionable binaries.

To reduce the overhead imposed by the essential feature engineering in traditional ML, some researchers have focused on deep learning approaches. Authors in Raff et al. (2017) used recurrent neural networks to analyze the first 300 bytes of the header of Windows PE files. Work in Raff et al. (2018a) utilizes convolutional layers within a neural network to extract information on Windows PE headers to determine binary intent. Authors in Vinayakumar et al. (2019) implemented what they call *Windows-Static-Brain-Droid,* which implements multiple architectures in

a voting scheme. The features for the architecture are both raw byte information and parsed features from the binary. The raw byte features feed into various architectures based on Raff et al. (2018a). The parsed features feed into multiple traditional classifiers and a FFNN. Section 3 will focus on developing an optimized neural architecture similar to Vinayakumar et al. (2019)'s FFNN.

2.1.2. Dynamic analysis

Unlike static analysis, dynamic analysis executes a binary to monitor its behavior. This is most often carried out in a sandboxed environment to restrict the binaries access to other resources which a malware could attack. Data collected from the execution behavior of malware allows for greater insight into a questionable binary's intent and nature. Authors in Fan et al. (2016); Luckett et al. (2016) utilized system calls captured during execution to detect malware. Work in Fan et al. (2016) utilizes traditional machine learning approaches, while Luckett et al. (2016) uses neural networks for classification. Authors in Tobiyama et al. (2016), look at API calls made in 5 minute intervals to classify binaries as benign or malicious. These calls were passed to a CNN for classification. In Huang and Stokes (2016), authors use FFFNs to classify binaries based on extracted API calls from dynamic execution.

Compared to static analysis, these methodologies require extra computational overhead and time to detect malware. However, dynamic analysis will not be able to detect sophisticated malware that can detect the presence of an emulation sandbox or the lack of network connectivity that is often found with isolated emulation environments.

2.1.3. Online analysis

Where dynamic analysis only analyzes the execution of a single binary, online analysis collects data from an entire system to monitor (in real time) for malware execution. This allows for continuous monitoring of an open system (not in a sandbox), with full access to all resources. Additionally, this allows for collection of execution details that extend beyond that of a single binary. This can include both knowledge of normal execution of a given system as well has effects to adjacent processes from live malware execution.

The authors in Demme et al. (2013); Ozsoy et al. (2015) utilize performance counters from an entire system to detect the presence of malware. Guan et al. (2012) proposed using system calls to detect malware in online systems with ensembles of Bayesian predictors and decision trees. Others have proposed using memory features (Xu et al., 2017). McDole et al. (2021) and Abdelsalam et al. (2019) show that per-process performance metrics from Ubuntu machines can provide high detection performance when ingested with a CNN. The process data is structured in the shape of an image, with the rows denoting different processes and the columns denoting different performance metrics for each process, collected from the target machine. Abdelsalam et al. achieves 89.5% detection accuracy using shallow CNNs, while McDole et al. achieves 92.9% detection accuracy using state-of-the-art CNNs on the same dataset. Kimmel et al. (2021) uses recurrent neural networks (RNNs) on the same dataset as McDole et al. and Abdelsalam et al., but organizes the inputs to the RNN as sequences of unique process features, all from the same time slice. They achieve 99.61% detection accuracy with this technique. Online malware detection can incur high overhead with continuous monitoring of systems, but provides real-time detection performance on evasive and low-lying malware in a live environment without requiring the identification of a suspicious executable.

2.2. Deep learning for malware detection

Using deep learning for malware detection has been researched extensively and spans approaches that utilize various types of deep learning algorithms including CNNs (Abdelsalam et al., 2018; Ganesh et al., 2017; McDole et al., 2020; Wang et al., 2019; Yeo et al., 2018), RNNs (Agrawal et al., 2019; Jha et al., 2020; Kimmel et al., 2021), feed forward neural networks (FFNNs) (Rudd et al., 2019; Sewak et al., 2018; Vinayakumar et al., 2019), etc. Deep learning approaches presented in these works have the advantage over traditional ML models as they do not require hand designed features in order to be performant. Although these approaches impose additional performance overhead as compared to some traditional ML algorithms, many have shown to be more performant under some conditions (Kamath et al., 2018), with high accuracy in malware detection (Kimmell et al., 2021).

Despite the fact that deep learning approaches have shown tremendous results for malware detection, most of these works fall short because either (1) they utilize state-of-the-art models that are not tailored specifically for malware detection and may not be optimal for the data available, or (2) they have hand designed their models specifically for malware detection, without AutoML, which requires extensive domain experts' knowledge and hand tuning. Fortunately, AutoML can help overcome these obstacles and attain higher optimal results; however, the feasibility of utilizing AutoML for malware detection is hardly explored.

2.3. AutoML overview

2.3.1. Neural architecture search

The performance of a model is highly dependent on the design of its architecture. A neural architecture search aims to find the architecture design that achieves the highest performance on unseen validation data. We consider a change in architecture design to constitute a change to the number or configuration of trainable parameters, or the layers' activation function.

2.3.2. One-shot search methodology

Many recent NAS methodologies focus on the computer vision domain. This field is heavily dependent on convolutional neural networks. Many types of layers within these networks, given the same shape of input, will produce the same shape of output. A network whose layers meet this condition is, intuitively, easily mutable; layers can be swapped out interchangeably, allowing the next layer to accept any chosen layer type's input since they share the same output shape as shown in Fig. 1. Many types of layers can be substituted for *Layer N* and maintain the same output shape of *(1, 16, 16)*. This property allows for an algorithm to test multiple layer choices at each layer of a network to find the best architecture configuration. The work presented in Pham et al. (2018) can create a *super-graph* which contains multiple *sub-graphs* representing all permutations of networks given the choices of each layer. A similar work, (Liu et al., 2018) relaxes the constraints of the categorical layer choice to a softmax choice, such that the categorical choice is now continuous, and a gradient can be used to find the best layer choices through training by backpropagation.

These NAS methodologies are used to learn an entire network architecture or learn the architecture of a cell that is repeated throughout the network. As long as each of the operations (layer) choices produce the same shape of output, the specific operation choices within a cell can be anything. This possibility allows for designing not only convolutional cells, but also recurrent cells. This allows the algorithm to find both CNNs and RNNs, or a combination of both.

These algorithms, known as One-Shot algorithms, find the most performant network configuration in "one-shot", without the need to train the network from scratch multiple times, by leveraging the output shape property.

2.3.3. Multi-trial search methodology

Multi-Trial NAS solutions, as opposed to one-shot, require many trials of different network configurations to find a performant architecture. In the past, before the invention of clever one-shot methodologies, this was the only way to test out different network architectures. Today, some types of networks still rely on multi-trial NAS, such as networks that can't easily swap out layers because of layer output shapes. There has been some work to improve the efficiency of this process, such as

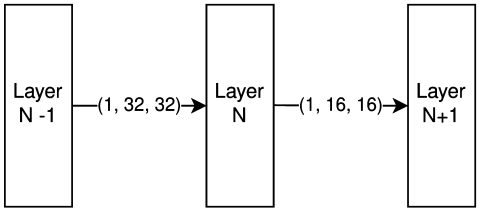

Fig. 1. Example Convolutional Layer Output Shapes.

Wei et al. (2016), through weight sharing. This allows each trial to run for a much shorter amount of time by leveraging the learned parameters from previous trials, and only optimizing for new parameters. However, these algorithms, if not carefully controlled, can become unstable in later trials. For this reason, our work with deep feed forward neural networks in Section 3 utilizes the more primitive multi-trial methodology in searching for the most performant architecture.

2.3.4. NAS search space

The NAS search space is the total space containing the values of all valid model configurations. During the NAS process, architecture configurations are drawn from this space and evaluated. The search space is arbitrarily large, so reasonable constraints are placed in order to bound the cost of time required to search and limit the complexity of the chosen model. For example, a model depth of 1,000 layers is a valid choice for an architecture, but it will produce a very complex model with a considerably high training time. For this reason, upper boundaries are usually provided. For example, in our proposed approach in Section 3, we set the number of layers' upper bound to 14 to limit the complexity of the model architectures available within the search space. Beside bounding the range of the search space, we also considered the sampling granularity and distribution of the search space. For example, in Section 3 we set the granularity in selecting a layer's width to 128 neurons in order to limit the number of available selections while still maintaining an appropriate level of expression of its effect on model performance. In order to simplify the NAS process, when a parameter value is selected from the search space, we fix this value throughout the model, instead of on a per-layer basis.

2.3.5. Automated ML for malware detection

Automated machine learning has recently been used in a variety of fields. Several AutoML works have been designed for specific domains, such as processes developed for the computer vision domain (Liu et al., 2018; Pham et al., 2018). However, AutoML is still at nascent stage which is yet to see wider adoption and application in cybersecurity.

The field of malware detection has barely seen the presence of AutoML, and to the best of our knowledge has only been presented in a few works. Research in Kundu et al. (2021) tested both AutoGluon-Tabular[1] and Microsoft NNI[2] on the EMBER-2018 dataset (Anderson and Roth, 2018), a malware dataset based on static analysis. These frameworks are used to tune hyper-parameters of a LightGBM model to best classify binaries from the dataset. Authors also used a proprietary dataset to evaluate the AutoML frameworks. This approach yielded a 3.2% increase of True Positive Rate above the EMBER-2018 baseline results with the same classifier. AutoGluon-Tabular produced these results vs a 2.2% increase with Microsoft NNI. Their approach largely used traditional machine learning methods as well as a FFNN in the ensemble offered with AutoGluon-Tabular. The authors of Isingizwe et al. (2021) use AutoML to detect malware from encrypted network traffic. They used TLS fields as parameters to form their AutoML process. This work

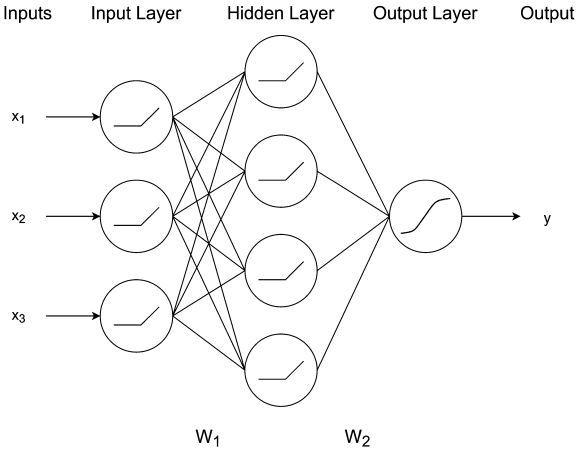

Fig. 2. Feed Forward Neural Network.

used a python package *mljar-supervised*,[3] utilizing many traditional ML models as well as a deep neural network in an ensemble.

3. Automated machine learning for static malware detection

3.1. Deep feed forward neural networks

Deep Feed Forward Neural Networks (FFNNs) are an extension of the simple perceptron network, except they often contain one or more hidden layers. FFNNs without convolutional or recurrent layers can also be referred to as Multi-Layer Perceptrons (MLPs).

FFNNs pass input data through each layer in the model sequentially, applying each layer's function to the previous layer's output, forming what can be seen as an acyclic graph from input to output with data flowing only one way. Fig. 2 shows an example FFNN. Each node within a layer can apply an activation function to the sum of each of its inputs, shown as the function lines within each node in the figure. Each connection between nodes has a specific weight applied to the output of a specific node into another node. W_1 and W_2 represent the set of weights between each layer, each weight in each set corresponding to a connection between two unique nodes. The network can have any number of hidden layers.

Deep FFNNs require backpropagation through gradient descent to train weights of each layer of the network sequentially, backward through the network, from output to input. Through the processes of backpropagation, activation functions such as sigmoid and tanh can lead to a problem called vanishing gradients. This occurs because repeatedly taking the gradient of these functions, as backpropagation occurs, results in a value that approaches zero. For this reason, idempotent activation functions such as Rectified Linear Unit (ReLU) are often used in hidden layers of deep networks to solve the vanishing gradient problem. Additionally, functions like ReLU and Exponential Linear Unit (ELU) are cheaper to compute than sigmoid and tanh, but still allow for the network to learn non-linear functions. However, sigmoid like functions allow for an output to be mapped to a probability, and are often used on the output layer of a network to allow for each output neuron to produce a binary decision. Fig. 2 shows the input and hidden layer activation functions as ReLU, and the output layer's activation function as sigmoid.

3.2. Search methodology

During the neural architecture search, an architecture selected from the search space is evaluated using an evaluation metric (F1-score our

[1] https://auto.gluon.ai/stable/index.html.
[2] https://github.com/Microsoft/nni.
[3] https://supervised.mljar.com/.

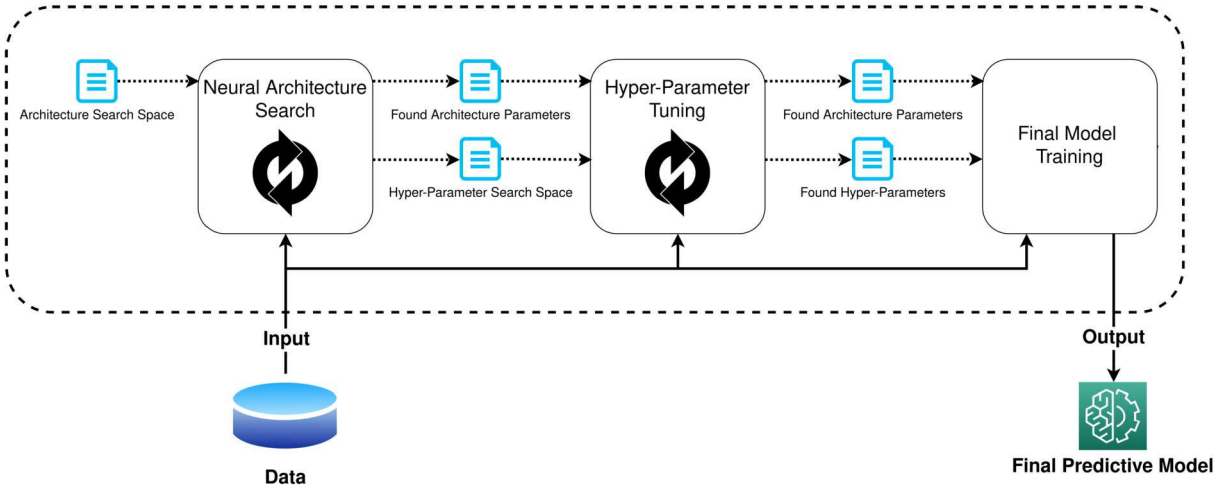

Fig. 3. Automated Machine Learning Process.

case), indicating the model performance based on which a strategy is employed to select the next architecture choice for subsequent evaluation. The next architecture selection in this work is based on a random selection strategy, where, regardless of the previous result, each new architecture choice is randomly selected without duplication. The NAS selects a number of random architecture configurations from the search space. These are referred to as trials. In each trial, a model is trained based on the selected architecture for a predefined number of epochs. Afterward, the model is evaluated at the end of every epoch using the evaluation score of the validation set. The model configuration that achieves the highest score will pass to the next phase, that is hyper-parameter tuning.

3.2.1. Hyper-parameter tuning

Once an architecture is selected during the NAS phase, the next phase, as shown in Fig. 3, searches for the optimal hyper-parameters of the chosen architecture. The hyper-parameters of the model are tune-able values that affect the model performance but do not alter the architecture of the model itself. This can include the batch size, optimizer, learning rate, dropout rate, etc. Just as in the NAS phase, the hyper-parameter tuning phase also has a bounded search space with a defined sampling granularity. With the hyper-parameter search space, we also define the sampling distribution. The hyper-parameter search phase uses the Tree-structured Parzen Estimator (TPE) strategy (Bergstra et al., 2011) in selecting the next set of hyper-parameters to test.

3.2.2. Final model selection

After the hyper-parameter tuning phase is complete, the results from the NAS phase and hyper-parameter tuning phase are combined to be the final model configuration. The model is then trained and evaluated after every epoch using the evaluation score of the validation set, and the highest performing epoch is saved as the final trained model to be evaluated on the test set.

3.3. Static malware data sources

We use two popular static malware datasets - EMBER-2018 (Anderson and Roth, 2018) and SOREL-20M (Harang and Rudd, 2020), extensively used in the literature. We use these datasets with more primitive AutoML to explore the results of this methodology on datasets that have had high result benchmarks set.

3.3.1. EMBER-2018 dataset (Anderson and Roth, 2018)

EMBER is considered to be the first attempt to create an appropriately large static malware dataset. The dataset contains features extracted from benign and malicious portable executable (PE) files using the 'Library to Instrument Executable Formats' (LIEF) (Thomas, 2017). The samples in the dataset are labeled as either malicious, benign, or unknown. Only the samples labeled malicious or benign are considered in this work. There are approximately 600 K samples in the training set and 200 K samples in the test set. Since there is no validation set provided, we excluded and used the last 20% of the training set (i.e. 120 K samples) as the validation set. There are two versions of the EMBER dataset: EMBER-2017 and EMBER-2018. EMBER-2018 was specifically curated so that the training and testing sets would be harder to classify. We used EMBER-2018 in our experiments. However, to fairly compare our results to other works that used EMBER-2017, we test and report our model's (found with EMBER-2018) performance against EMBER-2017 dataset.

3.3.2. SOREL-20M dataset (Harang and Rudd, 2020)

Sophos Labs[4] released SOREL-20M dataset in 2020 to address some shortcomings of EMBER dataset. This dataset contains 12,699,013 training samples, 2,495,822 validation samples, and 4,195,042 testing samples. SOREL-20M uses the same features from the EMBER-2018 dataset. The samples in the SOREL-20M dataset contain the same binary malicious label as EMBER-2018, but also contain extra metadata, including the number of anti-virus vendors that flagged a sample as malicious and the tags that anti-virus vendors associated with a sample. Included in these tags are labels such as *dropper*, *adware*, *downloader*, etc. Authors in Rudd et al. (2019) have shown that the use of this metadata can help to improve performance, and our work in this section will allow the possibility of a model to use this auxiliary information in the training process.

3.4. AutoML tuning and training

3.4.1. NAS phase configuration

The full architecture search space for both the EMBER-2018 and SOREL-20M experiments is shown in Table 1. The available options for *Activation* and *Tag Head Activation* are not applicable since the choices are either Rectified Linear Unit (ReLU) or Exponential Linear Unit (ELU). Similarly, for *Use Counts* and *Use Tags*, the choices are either *True* or *False*.

As mentioned previously, the SOREL-20M dataset has readily available labels each containing a binary malicious label, an encoding of the vendor tags, and a numerical count of the vendors flagging the sample

[4] https://www.sophos.com/en-us/labs.

Table 1
Architecture Search Space.

Parameter	*Minimum*	*Maximum*	*Granularity*
Depth	1	14	1
Width	128	1920	128
Activation	-	-	-
Tag Head Depth*	1	3	1
Tag Head Width*	16	112	16
Tag Head Activation*	-	-	-
Use Counts*	-	-	-
Use Tags*	-	-	-

*SOREL-20M Models Only

Table 2
Hyper Parameter Search Space.

Parameter	*Minimum*	*Maximum*	*Granularity*	*Distribution*
Batch Size (SOREL-20M)	128	16384	1024	quniform
Batch Size (EMBER)	32	8192	32	quniform
Learning Rate	0.0001	1.0	-	loguniform
Dropout	0.0	0.50	0.05	quniform
Tag Loss Weight*	0.0	1.0	.05	quniform

*SOREL-20M Models Only

as malicious. These additional labels were made available during the architecture search process through the use of additional output heads of the model to predict the count of the vendors flagging the sample malicious and predict any tags associated with the sample from anti-virus vendors. These additional heads were made optional through the use of two additional architecture search parameters: *Use Counts* and *Use Tags*, as shown in Table 1. Design for additional heads, their respective loss functions, and the network design is inspired by Rudd et al. (2019). The architecture search selects 150 random architecture configurations from the search space. The number of trials was chosen to cover both the search space and minimize cost. However, further investigation is required to analyze the effects of the number of trials on the selected models' performance as explained in subsection 3.6. The SOREL-20 and EMBER-2018 NAS was run for 10 and 25 epochs, respectively.

The highest achieved F1-score of a model during any point of its trial (instead of the F1-score of the final epoch) is chosen as the fitness score so that a model configuration's ability is more accurately represented, as the model's performance may fluctuate during the training process. Even though random search has been shown to give adequate results with a sufficient amount of trials (Bergstra and Bengio, 2012; Cheng, 2010), trial count remains a parameter to be investigated in future work.

3.4.2. Hyper-parameter tuning phase configuration

The full hyper-parameter search space is shown in Table 2. The *quniform* distribution behaves like the sampling granularity in the NAS phase. The *loguniform* samples from a logarithmic distribution such that the logarithm of the values returned will be uniformly distributed. Learning rate is sampled from this distribution to allow smaller values to be as likely sampled as larger values. We set the batch size minimum, maximum, and sampling granularity larger for the SOREL-20M experiments due to the size of the dataset as compared to EMBER-2018. Similar to the NAS phase parameters configuration, we believe that the minimum, maximum, and sampling granularity values requires further investigation.

In Rudd et al. (2019), using the SOREL-20M dataset, the authors use a loss weight of 0.1 for the vendor count head and vendor tag head, and a 1.0 loss weight for the malicious decision head. These loss weights can be considered a hyper-parameter available for tuning since altering the value does not change the architecture of the model. In our work, the malicious decision head loss weight is fixed to 1.0 while the auxiliary loss head weights are variable between 0.0 and 1.0 each. Note, only the tag head loss weight is included in Table 2 because the highest achieving model during the SOREL-20M NAS phase did not have a vendor count head, and therefore did not utilize that parameter. The models are trained for 10 and 25 epochs in the case of SOREL-20M and EMBER, respectively. F1-score is again used as the evaluation metric in selecting the highest performing model.

Table 3
Found Optimal Parameters.

Parameter	SOREL-20M	EMBER-2018
Depth	8	3
Width	1920	1664
Activation	ReLU	ReLU
Dropout	0.15	0.30
Learning Rate	0.000439	0.000269
Batch Size	3072	1440
Use Count Head	False	-
Use Tag Head	True	-
Tag Head Depth	1	-
Tag Head Width	112	-
Tag Head Activation	ELU	-
Tag Head Loss Weight	0.70	-

3.5. Experimental results

3.5.1. Evaluation metrics

We use four evaluation metrics along with receiver operating characteristic (ROC) and area under the curve (AUC).

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{1}$$

$$Precision = \frac{TP}{TP + FP} \tag{2}$$

$$Recall = \frac{TP}{TP + FN} \tag{3}$$

$$F1 - score = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{4}$$

Positive refers to a malicious sample, whereas, negative refers to a benign sample. *TP*, *FP*, *TN* and *FN* are true positives, false positives, true negatives and false negatives, respectively. Precision suffers when benign samples are labeled as malicious (high FP), while recall suffers when malicious samples are labeled as benign (high FN). F1-score is the harmonic mean of precision and recall, so it signifies models that have both high precision and recall. If a model has high precision but low recall or vice versa, the F1-score will be low.

3.5.2. Results

After the experiments, using F1-score as an evaluation metric at each phase of the process, the found architectures and hyper-parameters are shown in Table 3.

The detection results are listed in Table 4 and Table 5 for SOREL-20M and EMBER datasets, respectively. Also included in this table is the AUC with a maximum false positive rate (FPR) of 0.1%, the accuracy, F1-score, true positive rate (TPR) at 0.1% FPR, and TPR at 1% FPR. The table also contains results using loss as a performance metric for SOREL-20M and EMBER-2018; this is to show the difference in F1-score and loss as a performance metric in the final stage, this will be brought up in the discussion section of this section. Some other works shown in the tables only report a subset of the metrics, but are still shown for comparison.

In particular, for SOREL-20M, Table 4 shows our AUC results are on par with the FFNN ensemble from Nguyen et al. (2021) and slightly exceed Rudd et al. (2019), the work that presented the auxiliary model heads for SOREL-20M. Our model significantly exceeds the AUC under 0.1% FPR of the only other work (Nguyen et al., 2021), which reported this parameter. The accuracy of our model is similar but higher than Nguyen et al. (2021). We reported TPR at 0.1% and 1% FPR for comparison to Rudd et al. (2019), where it can be seen our model performed better in both.

Table 4
SOREL-20M Dataset Results.

Work	*Perf. Metric*	*AUC*	*AUC ≤ 0.1% FPR*	*Accuracy*	*F1-Score*	*TPR: 0.1% FPR*	*TPR: 1% FPR*
ALOHA Rudd et al. (2019)	-	0.997	-	-	-	0.922	0.972
FFNN Ensemble Nguyen et al. (2021)	-	**0.998**	0.0927	0.988	-	-	-
LightGBM Ensemble Nguyen et al. (2021)	-	0.984	0.0446	0.861	-	-	-
Our Work	F1-Score	**0.998**	0.966	**0.990**	**0.984**	**0.965**	0.993
Our Work	Loss	**0.998**	**0.969**	**0.990**	**0.984**	0.963	**0.995**

Table 5
EMBER Dataset Results.

EMBER 2018

Work	*Perf. Metric*	*AUC*	*AUC ≤ 0.1% FPR*	*Accuracy*	*F1-Score*	*TPR: 0.1% FPR*	*TPR: 1% FPR*
AutoGluon Ensemble Kundu et al. (2021)	-	-	-	-	-	**0.900**	-
Malconv w/ GCG Raff et al. (2020)	-	0.980	-	0.933	-	-	-
LightGBM Ensemble Nguyen et al. (2021)	-	0.986	0.0605	0.940	-	-	-
Detection Pipeline Loi et al. (2021)	-	**0.995**	-	**0.969**	-	-	-
Our Work	F1-Score	0.984	**0.614**	0.958	**0.958**	0.417	**0.969**
Our Work	Loss	0.981	0.573	0.918	0.921	0.188	0.951

EMBER 2017

Work	*Perf. Metric*	*AUC*	*AUC ≤ 0.1% FPR*	*Accuracy*	*F1-Score*	*TPR: 0.1% FPR*	*TPR: 1% FPR*
DeepMalNet Vinayakumar et al. (2019)	-	-	-	0.989	0.989	-	-
MalConv Raff et al. (2018a)	-	-	-	0.988	0.988	-	-
Our Work	F1-Score	**0.999**	**0.916**	**0.992**	**0.992**	**0.956**	**0.997**

With respect to EMBER-2018 in Table 5, Loi et al. (2021) performs slightly better in their reported metrics, AUC and accuracy, whereas the rest of their metrics are not reported. Our model chosen in the final training phase is similar to other results in AUC and accuracy, surpassing (Raff et al., 2020) in AUC, and surpassing both (Nguyen et al., 2021; Raff et al., 2020) in accuracy. The AUC under 0.1% FPR of our model far surpasses the results of Nguyen et al. (2021). The TPR at 0.1% is the only reported metric of Kundu et al. (2021), which is significantly higher than our results. Due to limited metrics provided by other related works, it is difficult to compare the efficacy of our AutoML method in a holistic sense. The results from EMBER-2017 (with the optimal parameters from EMBER-2018) are reported in the bottom of Table 5. The authors in Vinayakumar et al. (2019) and Raff et al. (2018a) only reported accuracy and F1-score of their results. Our model's accuracy and F1-score are slightly higher than their results, but with metrics close to 100%, this is significant.

The results show that models developed with our proposed AutoML pipeline are similar to those found with hand designed solutions, and sometimes even exceed the performance. This shows the efficacy of integrating AutoML into a malware detection pipeline, eliminating the need to hand designed models, which is difficult, time consuming, and requires high technical skills.

Fig. 4 shows the ROC curve for both the EMBER-2018 and SOREL-20M experiments, note the logarithmic scale on the x-axis denoting the FPR. An ROC curve shows the TPR for each respective FPR. Given the magnitude increase of training data in SOREL-20M over EMBER-2018, it is no surprise the TPR of SOREL-20M is higher than EMBER-2018 at any FPR. SOREL-20M TPR falls off much slower than EMBER-2018, and never goes below 0.8 TPR in the graph.

The training and evaluation were performed on a virtual machine equipped with 92 vCPUs and 448 GB of memory. Additionally, it incorporated 8 Tesla V100 GPUs, each with 16 GB of VRAM. While the number of cores and memory might have been excessive for the task at hand, the presence of 8 GPUs facilitated parallel training sessions for neural network hyperparameter optimization.

3.6. Discussion and analysis

3.6.1. Meta-hyper-parameter selection

As mentioned earlier, many of the parameters governing the NAS and Hyper-Parameter tuning phases are selected based on our experience to simplify the process and provide a balance between the cost of training and detection results. We discuss below some of these parameters.

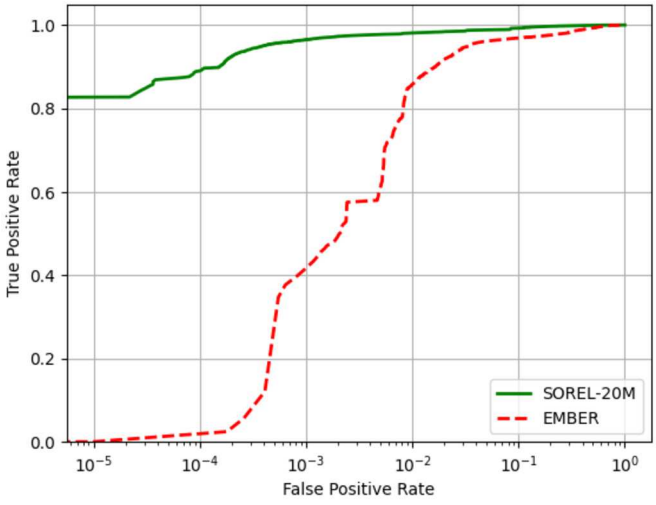

Fig. 4. ROC using F1-Score for Selection.

3.6.2. Epochs per trial

The number of epochs per trial is an important parameter that directly affect the NAS process. This was especially a consideration for the SOREL-20M trials, since the dataset is an order of magnitude larger than the EMBER-2018 dataset and therefore took much longer to train.

Initially, the number of epochs for the SOREL-20M NAS trials was set to 3. This implies that the model configurations with the highest performance after training for 3 epochs would perform the best overall. To test this, we increased the number of epochs to 10 and 20 to help in better understanding of the impact of epochs per trial on the selected models' performance during the NAS. The results of these experiments are shown in Fig. 5. The primary consideration here is with the performance trend of SOREL-20M, but EMBER-2018 is shown as well.

This graph shows the F1-score average of the top 30 selected models at any given epoch. The F1-score for each model is calculated as the highest F1-score reached up to and including a given epoch. At each

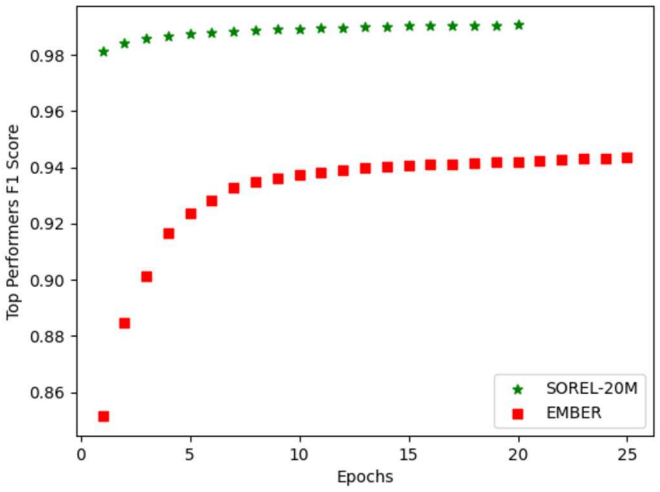

Fig. 5. Top 30 Preforming Models Average F1 by Epoch.

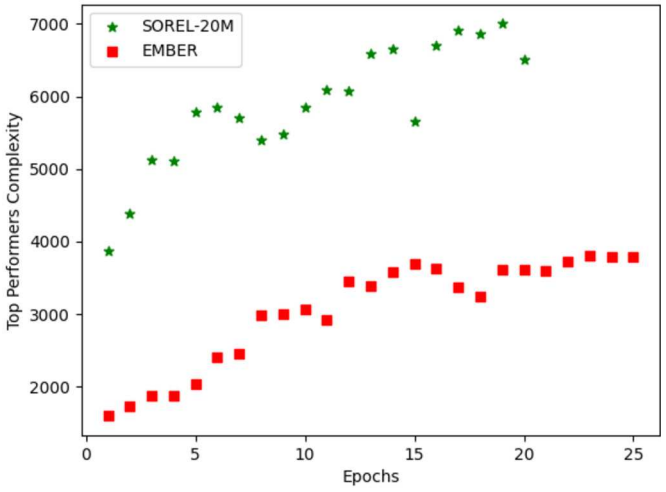

Fig. 6. Top 30 Preforming Models Average Complexity by Epoch.

epoch, the 30 models with the highest F1-score, as described above, are averaged together. At any epoch, the top 30 set of models may be different if any model in the experiment achieves results that puts the model in the top 30 for that epoch. It can be seen that there is a correlation between top model performance and the number of epochs, in a seemingly logarithmic relationship. As long as a model is not so complex that it over-fits the training data, a more complex model should, intuitively, preform as well as or better than a less complex model. However, a good choice for the number of epochs should be where the curve start to straighten so that the model doesn't become too complex and, in turn, require a massive amount of training time. Fig. 6 shows that as the number of training epochs per trial increases so does the average complexity of the top 30 performing models. The model complexity in the figure represents the average product of *width* and *depth* of the hidden layers of the top 30 model configurations during the NAS phase, which results in the number of trainable parameters in a given model.

3.6.3. NAS and tuning-parameters phases evaluation metric

We use F1-score as an evaluation metric to select the models that have both high recall and precision. After getting the final selected model during the NAS and hyper-tunning-parameter phases, we train and evaluate the model using both F1-score and binary cross-entropy loss. The results shown in the ROC curves in Fig. 7 shows that both

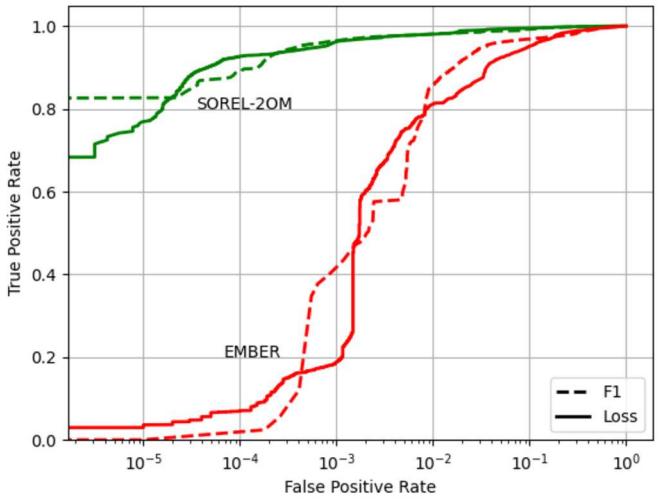

Fig. 7. ROC using Loss vs F1 for Selection.

metrics can reach comparable results. This indicates that, besides F1-score, other metrics could also be explored, including accuracy, AOC, binary cross-entropy loss, etc. and are left to future work.

3.6.4. Search space bounding and strategy

The search space values are one of the most important factors in the AutoML process. As shown in Table 3, the width parameter (i.e. 1664) of selected model for EMBER-2018 dataset is found to be less than the maximum value (i.e. 1920). However, the width of the SOREL-20M model was the maximum available value in the bounded search space. This indicates that an even wider model might preform better than the found model had the search space been bigger. Selecting optimal search space values is still an open question. The choice of the search strategy for the NAS and tuning-parameter phases is random search and TPE, respectively. In this section, these strategies were chosen because of their simplicity. However, a more adequate strategy tailored to malware detection could potentially result in better selected AutoML models.

3.6.5. Cost of current implementation

The SOREL-20M experiments took ≈30 minutes per epoch to run, with 16 experiments running simultaneously. The EMBER-2018 experiments took ≈5 minutes per epoch, with 24 experiments running simultaneously. Overall, the experiments took ≈5600 minutes and ≈1560 minutes to run both SOREL-20M and EMBER-2018 experiments, respectively. This is the time to run the NAS and hyper-parameter phases of the process, excluding the final model training. The time to train the final model is not reported, as the computational cost is insignificant compared to the previous two phases.

The current implementation of the proposed methodology uses a *multi-trial NAS*, where each set of model parameters selected from the NAS search space are trained to the specified epoch limit. Other implementations of multi-trial NAS try to optimize this process through early stopping and weight sharing (Li and Talwalkar, 2020). Even though these methods may introduce instability into the process, they can reduce the computational cost.

It can be concluded that it is more expensive to use AutoML than to train hand-designed models. This cost trade-off should be taken into account as the proposed methodology becomes more refined. Future implementations may significantly reduce the time to complete the AutoML process. This can be achieved through more sophisticated NAS implementations and intelligent search strategies that can reduce the number of trials required or the number of epochs required per trial.

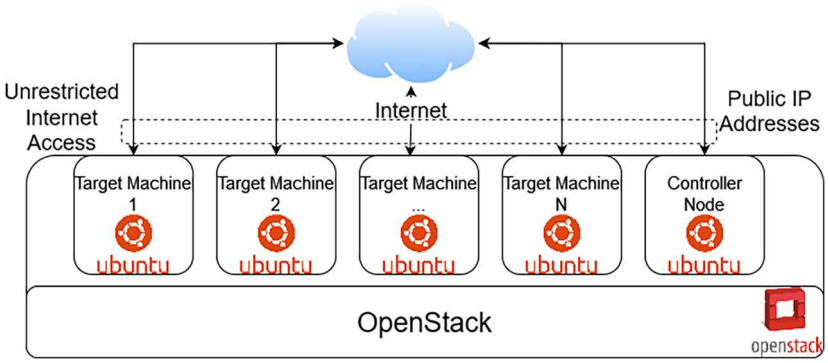

Fig. 8. Cloud Testbed Setup.

Fig. 9. Experiment Phases.

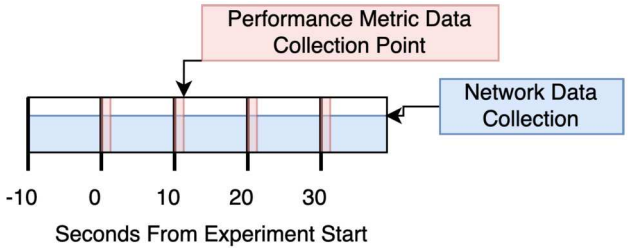

Fig. 10. Data Collection Phases.

4. Automated machine learning for online malware detection

This section will focus on using one-shot AutoML for malware detection in online cloud environments using Convolutional Neural Networks (CNNs).

4.1. Convolutional neural networks

CNNs are a widely used type of deep learning model designed for image type data. CNNs work differently than regular deep FFNNs, where the output of every node is passed into every node of the next layer. CNNs receive a 3 dimensional input (channels, height, width). CNNs have filters, whose values are learned, that convolve across input channels to detect edges. The primitive edges detected in earlier layers can be combined in later layers to learn more complex shapes. The core of CNN layers falls into two categories: normal and reduction convolutional layers. Normal convolutional layers use filters to convolve across the input to produce data with more channels, keeping the same height and width. Reduction cells to reduce the width and height of its input data to reduce the number of trainable parameters in the next layer or cell. The output of the convolutional layers is passed through a pooling layer to reduce the input dimension to 1 for dense layers to produce the network output (prediction).

CNNs are used in this section because process performance metric data can be grouped together in the form of an image, with rows denoting unique processes and columns denoting performance features of these processes.

4.2. Online cloud testbed

Fig. 8 illustrate the testbed utilized to generate the online malware dataset in an OpenStack[5] instance hosted by the University of Texas at San Antonio. All virtual machines used to create this dataset had open and unrestricted internet access, as well as a public IP address. Each virtual machine is running a fully up-to-date Ubuntu 18.04 instance. The experiments are controlled and data gathered by a controller node within the OpenStack testbed. Each VM contains programs to collect data from their respective sources, which at the end of the experiment is collected by the controller node. Before each experiment, each target VM is reset to a clean state. Each virtual machine has 2 CPU cores, 4 GB of RAM, and 40 GB of disk space.

4.3. Application and baseline sets

To best understand the behavior of malware on a full, online system, it may help to include malware data when the machine is idle and fully operational. For the purposes of this dataset, the fully operational server will be an Apache web server hosting a WordPress application, with a MySQL database on the backend. To model real world end users of the server, an on ON/OFF Pareto distribution following NS2[6] parameters is utilized to mimic the distribution of client requests to the webserver. All malware was run with only user level privileges.

4.3.1. Malware source and selection

The malware selected for this data came from a variety of sources, including VirusTotal,[7] MalShare,[8] VirusShare,[9] Linux-Malware-Samples,[10] and MalwareBazarr.[11] The gathered samples were tested for ability to execute on the target hardware in case the mutable header field of the malware had been altered, in which case the malware may not run on the target hardware. Also, samples that lead to corruption of the collected data during the experimentation process were removed from consideration after the fact. In total 4077 malware samples were considered.

4.3.2. Data collection

The experiment length for this dataset is 10 minutes - meaning data is collected for the entirety of 10 minutes. Halfway through a given experiment, the malware being tested is executed. Therefore, every experiment contains an equal amount of benign and malicious activity.

[5] https://www.openstack.org/.

[6] http://www.isi.edu/nsnam/ns/doc/node509.html.

[7] https://www.virustotal.com/.

[8] https://www.malshare.com/.

[9] https://virusshare.com/.

[10] https://github.com/MalwareSamples/Linux-Malware-Samples.

[11] https://bazaar.abuse.ch/.

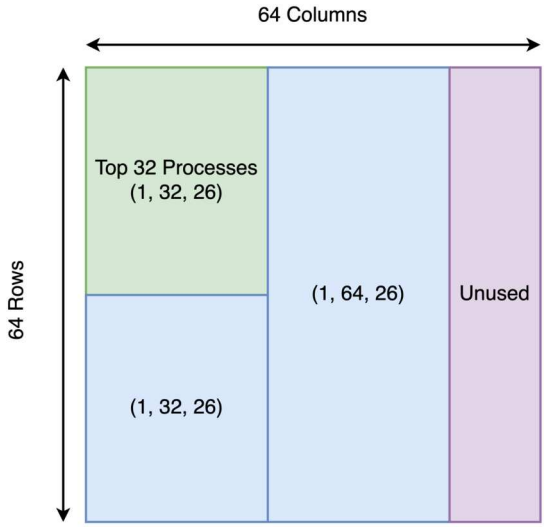

Fig. 11. Input Data Shape.

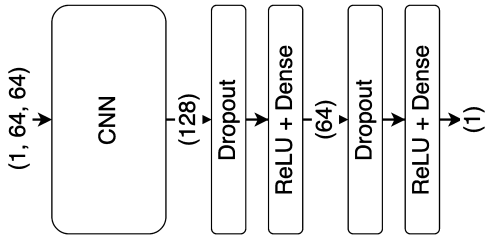

Fig. 12. General Architecture.

This can be seen in Fig. 9. There are multiple random benign SSH connections made to each target box throughout the experiment to mask the SSH connection used to spawn the malware execution.

The methods by which different sources of data are collected contain both continuous and discrete collection. Network data is collected continuously throughout the experiment, and starts 10 s early to allow for a delta to be taken, since the collection is a running total of network activity per process. Per-process data is collected at an interval of every 10 s, taking the instantaneous value of the monitored metrics. The collection over time for each data source as shown in Fig. 10. Specifics of each type of data collected will be discussed in the following subsections.

4.3.3. Per-processes performance data

Performance metrics are collected on a per-process basis. This data is collected every 10 s for the duration of the experiment. The python library `psutil` is used to collect this data.

Process IDs (PIDs) would, at first, seem like an easy way to identify a unique process thought the experiment, but this doesn't hold true. A Linux kernel by default has a maximum PID of 32768, at which point PIDs begin getting re-used. Therefore, it is feasible that in a highly active system that creates many new processes and closes old ones, that a single PID may identify more than one process during the experiment run-time. Instead, a tuple of the entire command line (including arguments) of the process and a hash of the executable (if applicable) is collected. This is much less likely to collide with the identifier of another process.

4.3.4. Per-process network data

Many data collection tools do not allow for the collection of network traffic statistics in a per-process basis. However, the tool Nethogs[12] allows for the grouping of bandwidth by process, and is used to collect network bandwidth data in the experiments. A python wrapper is used to interact with the Nethogs library for data collection.

The network bandwidth data (bytes in/out) per process is recorded as a running total, therefore network data collection is started 10 s early, and the delta between each record is used in post-processing. In order to match network data to process data, the PID at a given timestamp in the network data can be compared to the records in the process data, which ultimately holds the primary key to denote a unique process.

4.3.5. Combined data and representation

In order to include network data with per-process performance metrics, the data is combined. First, any record of the data collection agents is removed from the per-process performance data. The data that is left in process data will be the basis by which network usage is searched in network data. The discrete process data is grouped by collection time (every 10 s), and any matching network data between collection times is added to the latter process data collection record. That is for a given unique process record p taken at collection time N, any matching network data records for p between the previous collection time N-1 and current collection time N, will be added to the process record of p at collection time N. A sample feature table for a unique process is shown here in Table 6.

To feed the data to models, the data is represented as a single channel (grayscale) image. The columns of this image are the collected performance metrics and the rows are unique processes. As shown in Fig. 11, the image dimensions are represented as (channels, rows, columns) and are selected to be (1, 64, 64). The first 26 columns and second 26 columns each contain performance metrics for the rows of processes. That is, the 56 used columns of the input are divided into two sections of 26 features, each of these meta-columns representing a unique processes features. The first 32 rows of the first meta-column are reserved for commonly occurring processes found in the training set, so that in every input sample, a process that is commonly occurring will be in the same spot in the data in every input sample.

There are 12 blank columns, padded with 0, on the right side of the image that are used as padding so the image can maintain a square shape. The image shape is selected to be square and a power of 2 to ensure there are no dimensionality problems when feeding the data into a variety of CNN models. A total of 128 unique processes can be included in an image, and the top 32 processes that occur very frequently throughout the data will always be placed in the same row and column throughout all samples.

4.4. Methodology

We used one-shot learning to find a performant CNN to detect malware from the performance metric data. The Darts (Liu et al., 2018) AutoML methodology is applied to search for an optimal CNN architecture from the training data. The code for this is adapted from the Microsoft NNI implementation of Darts. Darts works to find normal and reduction convolutional cells by figuring out layer connections between nodes in the repeated cells. The found architecture will be a normal and reduction convolutional layer in a CNN with a specified number of layers (cells), nodes per cell, and channels per node. Increasing the number of nodes, and even more so increasing the number channels per node, can create large memory overhead in the neural architecture search. Darts finds the connections between nodes in a cell by posing the probability of a connection being the best as a softmax, so the best connections can be found using gradient descent. For further explanation on the DARTS AutoML process, refer to the original paper Liu et al. (2018).

The choices for connections between nodes in a cell are *skip connect* (identity for normal cells and factorized reduction for reduction cells),

[12] https://github.com/raboof/nethogs.

Table 6
Features Sample.

Metric	Value	Metric	Value	Metric	Value
num_fds	78	cpu_percent	0.0	cpu_time_user	0.15
cpu_time_system	1.7	cpu_time_children_user	7.64	cpu_time_children_system	3.1
context_switches_voluntary	1390	context_switches_involuntary	430	num_threads	1
memory_info_rss	9113600	memory_info_vms	163598336	memory_info_shared	6795264
memory_info_text	1376256	memory_info_lib	0	memory_info_data	18956288
memory_info_dirty	0	memory_info_pss	2922496	memory_info_swap	0
io_read_count	53242	io_write_count	18782	io_read_bytes	320275456
io_write_bytes	113713152	io_read_chars	248760749	io_write_chars	152977520
sent_bytes	0.0	recv_bytes	0.0		

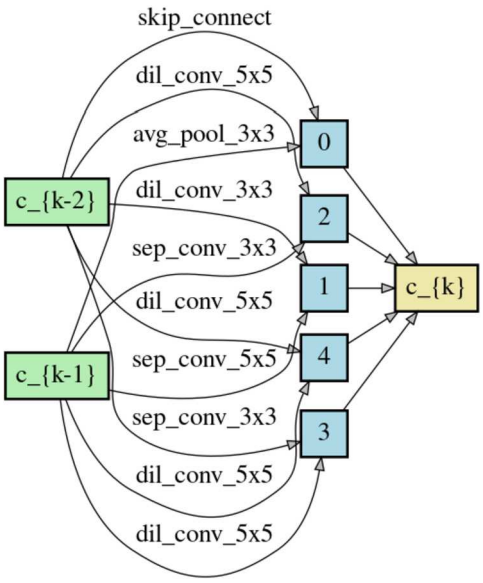

Fig. 13. Found Normal Cell.

dilated convolution (5x5 or 3x3), *separable convolution* (3x3 or 5x5), *average pooling* (3x3), or *max pooling* (3x3). These were the choices in the original Darts paper and are also used here. Stochastic Gradient Descent (SGD) optimizer and a learning rate scheduler are both used, with the same parameters as described in Liu et al. (2018).

The general architecture for the entire network is shown in Fig. 12. The CNN part of the model is either be found by with the Darts methodology or is a state-of-the art CNN for comparison.

4.5. Training and results

4.5.1. Data splits

Given 4077 total malware experiments per set (baseline/application), running for 10 minutes each, with data points at every 10 s, 246,620 total samples are available in the baseline and application dataset. 80% of the experiments are used for training, 10% for validation, and 10% for testing. No experiment (malware sample) is contained in more than one set (training/validation/testing). Also, the baseline training set consists of the same malware as the application training set, and the same is true for validation and test sets. A mean and standard deviation are calculated using the training set in the baseline and application set, and is used to normalize each of the respective datasets.

4.5.2. Neural architecture search

The Darts network for the baseline data is found, with the meta network parameters set at 5 layers, 5 nodes per cell, and 5 channels per node. Due to a performance decrease when the same Darts parameters are applied to the application dataset, the Darts model for the application set is fixed at 7 layers, 5 nodes per cell, and 9 channels per node. These choices are somewhat arbitrary, but have direct impact on memory usage during the NAS and the complexity and predictive performance of the found architecture. The selections made are to allow the model to fit on a single GPU while achieving good predictive performance. The impact of these choices is discussed (Liu et al., 2018).

A dropout rate of 0.30 is used in the neural architecture search, the same as used in all the rest of the model training. The Darts architecture search is run for 30 epochs (approximately 13 hours), using the training data. A batch size of 96 is used, the same as the original Darts paper. The found architecture is then trained using the same hyper parameters as the models it is compared to, described next.

4.5.3. Training parameters

In order to compare the performance of Darts to state-of-the-art CNNs, these models will be trained the same way as the found Darts models: *Resnet18*, *Resnet50*, *Resnet101*, *Densenet121*, *Densenet169*, and *Densenet201*. All the considered models share the same hyper parameters. The models are each trained for 100 epochs, use the Adam optimizer with a learning rate of 0.0005, learn on a batch size of 512, and have a dropout rate of 0.30. For each model, the epoch with the lowest validation loss is used on the test set to produce the final results for that model.

4.5.4. Results

The best found normal and reduction convolutional cells structures in the baseline darts model are shown in Figs. 13 and 14, respectively. The two input nodes in each cell are the outputs of the previous two cells, or in the case of the first cell the duplicated output of the first layer of Darts. All the node outputs are concatenated to be the cell output.

The training and evaluation was conducted on a VM with 14 vCPUs, 100 GB of memory, and an RTX A6000 GPU with 48 GB of VRAM.

The predictive results of the test set are shown in Table 7. This table shows the accuracy, precision, recall, F1-score, and Area Under the Curve (AUC) for each model. Additionally, to model a real world scenario, a threshold is calculated from the validation set, such that the validation false positive rate is 1.00%. This models a scenario where many false positives can become overwhelming for analysts to deal with, so an effort is made to minimize them by increasing the detection threshold of the malware detection model. When the threshold is increased, this can create a delay in a positive malware detection, in real time, through false negatives at the beginning of malware execution. This is shown in the table as *Delay @ Low FPR*, and is the average number of seconds elapsed before a successful detection after the malware injection point. Also shown in this section of the table is True Positive Rate *TPR* and False Positive Rate *FPR* at the high detection threshold (low FPR) on the test set.

Both of the Darts models that were tried are shown in the Application section of Table 7. The first model has 5 layers, 5 nodes per cell, and 7 channels per node. The second Darts model has 7 layers, 5 nodes per cell, and 9 channels per node. The Darts models in both baseline and application datasets perform better in almost every area than state-of-the-art models. In the baseline set, Resnet18 and Resnet50 show better precision and recall than the Darts model, respectively. It can, however,

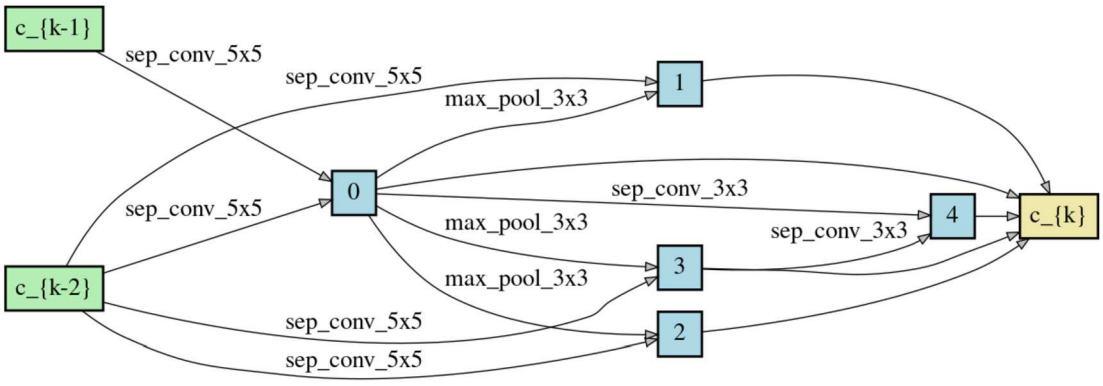

Fig. 14. Found Reduction Cell.

Table 7
Online Detection Results.

Baseline

Model	*Accuracy*	*Precision*	*Recall*	*F1-Score*	*AUC*	*Delay @ Low FPR*	*TPR @ Low FPR*	*FPR @ Low FPR*
Resnet18	0.97463	**0.99387**	0.95511	0.97411	0.99877	10.56373 s	0.96321	0.00735
Resnet50	0.97913	0.96266	**0.99689**	0.97948	0.99892	9.60784 s	0.96681	0.00759
Resnet101	0.98897	0.98856	0.98937	0.98897	0.99927	3.60294 s	0.98814	0.01045
Densenet121	0.98358	0.99079	0.97621	0.98344	0.99896	7.81863 s	0.97367	0.00816
Densenet169	0.98346	0.97972	0.98733	0.98351	0.99838	6.66667 s	0.97490	0.01086
Densenet201	0.98570	0.99148	0.97981	0.98561	0.99907	5.90686 s	0.98005	**0.00898**
Darts AutoML	**0.98917**	0.98674	0.99166	**0.98919**	**0.99954**	**3.03922 s**	**0.98986**	0.01094

Application

Model	*Accuracy*	*Precision*	*Recall*	*F1-Score*	*AUC*	*Delay @ Low FPR*	*TPR @ Low FPR*	*FPR @ Low FPR*
Resnet18	0.96246	0.94401	0.98709	0.96507	0.99417	21.33995 s	0.92964	0.01945
Resnet50	0.96667	0.96196	0.97512	0.96850	0.99480	14.54094 s	0.94084	0.01541
Resnet101	0.97953	0.97627	0.98500	0.98061	0.99728	10.19851 s	0.96470	0.01446
Densenet121	0.97239	0.97116	0.97644	0.97379	0.99414	13.44913 s	0.95592	0.01937
Densenet169	0.96164	0.94393	0.98554	0.96429	0.99248	28.31266 s	0.90368	0.01386
Densenet201	0.96078	0.94631	0.98103	0.96336	0.99276	27.89082 s	0.90671	0.01558
Darts AutoML (5 Layer)	0.97672	0.97755	0.97815	0.97785	0.99659	13.15136 s	0.95623	**0.01171**
Darts AutoML (7 Layer)	**0.98611**	**0.98520**	**0.98842**	**0.98681**	**0.99907**	**4.01985 s**	**0.98694**	0.01532

be seen that the Darts model has a higher F1-score signifying that the Darts model better balances precision and recall on the test set better than either of the other models. The Darts model also has the lowest delay, and is under 10 s, meaning that most of the malware in each execution experiment was detected in the first time slice after injection. Additionally, many of the state-of-the-art models are shown to impose a significant delay in the detection of the malware, with some averaging over 2 time slices, or over 20 s for a successful detection. The Darts models don't always have the lowest FPR at the high detection threshold, but all results in this column are shown to be close to the 1% target to validate the delay and TPR results.

Accuracy, *Precision*, *Recall*, and *F1-Score* are shown for each model in both sets in Figs. 15 and 16. The average malicious prediction delay is also shown in Figs. 17 and 18. The higher performance difference between the Darts models and state-of-the-art models in the application set vs the baseline set, suggests the need for AutoML derived models as data becomes more complex. Data from a server during real world use is more noisy and allows for malware execution to better hide within this noise. The neural architectures that are specifically derived based on this more complex data for this use case are more performant at identifying malware execution than generic architectures.

5. Future work and conclusion

5.1. Future work

This work describes the usefulness of AutoML for malware detection. Future works can expand on the ideas of this work with different search algorithms and malware data sources, as well as create tools to even further automate the process to make layman use of these methodologies easier.

5.1.1. Recurrent neural networks

Recurrent Neural Networks have shown near perfect results with online per-process performance metric data (Kimmel et al., 2021). The Darts methodology can also be used to derive recurrent cells, and this methodology should be examined on the dataset from Section 4 in the future.

5.1.2. Per-layer granularity

In our work in Section 3, once the width of the hidden layer is selected from the search space, it is fixed throughout the hidden layers of a model leading to a rectangular shape of the hidden layers in the model. However, an equivalent or a more optimal model may contain variable size layers with potentially fewer trainable parameters. A NAS process that allowed this level of granularity without an explosion of the NAS search space would prove valuable.

5.1.3. Refinement of meta-hyper-parameters

The set values of the meta-hyper-parameters have a significant effect on the efficacy of the AutoML process. Works such as Feurer and Hutter (2018) have developed methods to optimize a search strategy within the given confines of the meta-hyper-parameters in a data driven way. Finding the appropriate bounds of these parameters, specifically tailored to the malware detection domain, is yet to be explored.

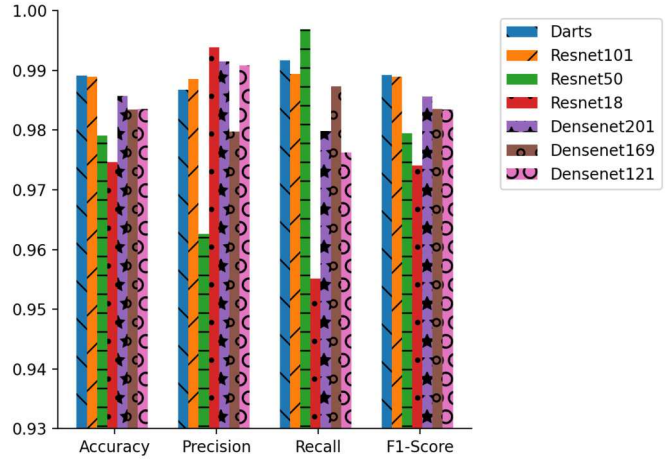

Fig. 15. Baseline Results. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

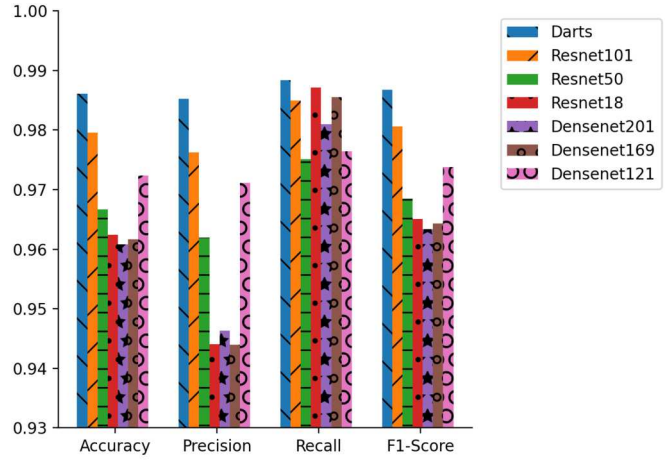

Fig. 16. Application Results.

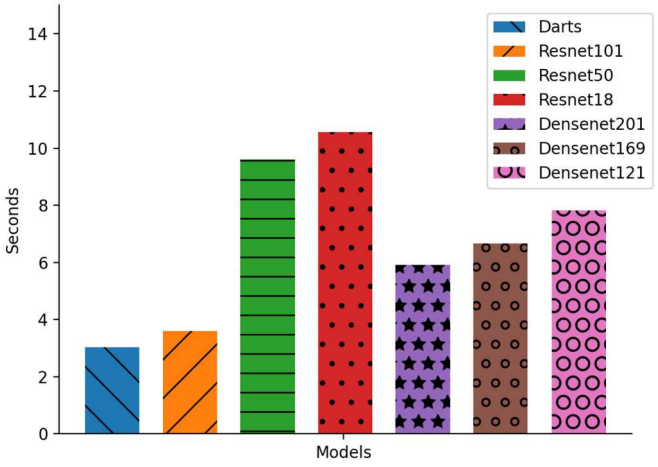

Fig. 17. Baseline Delay.

Addition of auxiliary output heads to the NAS search space can also be considered meta-hyper-parameters. One of the potential labels that can be given to this data may not be of use in a strictly detection setting, but may help derive a more performant model for the required objective with auxiliary loss, just as discussed in Rudd et al. (2019). Automatic inclusion of these in the search space based on label data would be valuable in automatic model searching. If hyper-parameter

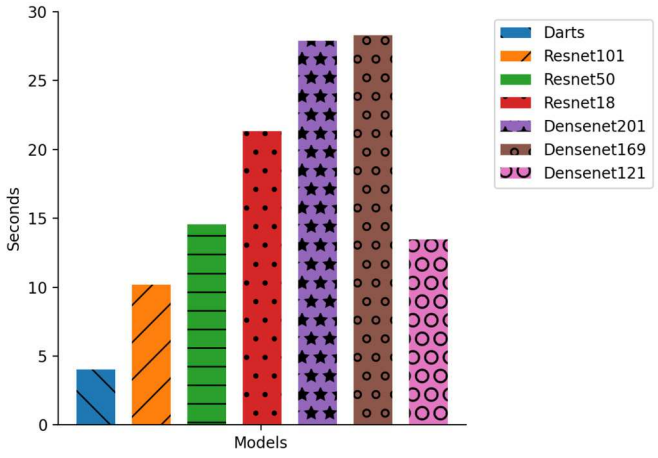

Fig. 18. Application Delay.

tuning is also performed as part of the AutoML process, the tuning algorithm can also be considered a meta-hyper-parameter. Depending on the evaluation metric, or rather intended performance (low false positive rate, high accuracy, etc), the found optimal parameters may differ. Algorithms such as differentiable evolution can also allow for optimization for multiple objectives (evaluation metrics).

5.1.4. Deep learning types and ensemble learning

In Section 3, we only used FFNNs for the SOREL-20M and EMBER-2018 datasets. An analysis of using various deep learning models can be very useful. Further, malware data can be extracted in many forms and types of data (e.g., time series and image data). Training a machine learning model on combined dynamic time series data and statically extracted tabular data can enhance the model's detection ability. However, designing such a model can be very difficult and, as such, AutoML is the perfect candidate for this task. An AutoML system that can intelligently conform to other sources of heterogeneous data is an area for future work.

In addition, AutoML can be utilized for ensemble learning. For instance, an AutoML system that can train multiple sub-models of different types and ensemble the sentiment of the sub-models would allow for more robust application in practice. Works such as Erickson et al. (2020) ensemble many types of machine learning models, including FFNNS, to achieve better results. Extending this to other deep learning model types could prove beneficial for malware detection.

5.1.5. User friendly AutoML

Designing AutoML models can be easier than designing a deep learning model from scratch, but an even more automated deep learning approach would be helpful for those with knowledge of their own data, but not necessarily deep learning. An AutoML system that could be instantiated with only training data inputs, type of data (vector, image, time-series), and primary and auxiliary labels would allow even broader access to malware detection solutions using deep learning. This framework would be able to automatically select a model type of deep learning architectures and use AutoML techniques to find a performant architecture to suit the data, making maximal use of any provided auxiliary information. Ideally, this would combine the methodologies and discussions from both Sections 3 and 4. It would perform all phases of the AutoML process efficiently, and be able to set applicable meta-hyper-parameters from details of the provided training data.

5.2. Conclusion

In conclusion, we conjecture that Automated Machine Learning offers an effective solution for detecting malware in both static and online cloud IaaS environments. We found that AutoML generated models

can perform just as well or even better than state-of-the-art models or models that have been handcrafted by experts with domain knowledge in machine learning and malware. We explored the performance of AutoML on two popular datasets static malware datasets in Section 3, SOREL-20M used to demonstrate efficacy on large datasets; and EMBER-2018, a dataset that was specifically curated to hinder the performance of machine learning models; with results in Tables 4 and 5. Our work on static malware datasets showed the feasibility of using AutoML as a tool for malware detection while reducing the external complexity and expertise required to train DL models.

We further explored one-shot AutoML on a new online cloud IaaS malware dataset using CNNs. Our results show that AutoML approaches can be utilized by cloud service providers and malware detection vendors to find custom deep learning models for malware detection utilizing any of a variety of data sources. The online approach we have shown can derive a custom CNN that is more capable than state-of-the-art models and contains cells that are more complex than what can feasibly be derived by hand. Importantly, we demonstrated that the difference in detection ability between AutoML models and state-of-the-art models becomes more pronounced as the noise in the input data increases, approaching the noise levels seen in real-world applications. We also elaborate on future directions to mature the use of AutoML research towards cybersecurity domains.

CRediT authorship contribution statement

Austin Brown: Conceptualization, Data curation, Methodology, Software, Writing – original draft. **Maanak Gupta:** Investigation, Supervision, Validation, Writing – review & editing. **Mahmoud Abdelsalam:** Investigation, Supervision, Visualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The authors do not have permission to share data.

Acknowledgements

This work is partially funded by the National Science Foundation grants 2230609, 2043324 at Tennessee Tech University, and 2230610 at North Carolina A&T State University.

References

Abdelsalam, M., Krishnan, R., Huang, Y., Sandhu, R., 2018. Malware detection in cloud infrastructures using convolutional neural networks. In: IEEE Conference on Cloud Computing, pp. 162–169.

Abdelsalam, M., Krishnan, R., Sandhu, R., 2019. Online malware detection in cloud auto-scaling systems using shallow convolutional neural networks. In: IFIP Annual Conference on Data and Applications Security and Privacy. Springer, pp. 381–397.

Abou-Assaleh, T., Cercone, N., Kešelj, V., Sweidan, R., 2004. N-gram-based detection of new malicious code. In: International Computer Software and Applications Conference, vol. 2.

Agrawal, R., Stokes, J.W., Selvaraj, K., Marinescu, M., 2019. Attention in recurrent neural networks for ransomware detection. In: IEEE Conference on Acoustics, Speech and Signal Processing.

Alotaibi, A., 2019. Identifying malicious software using deep residual long-short term memory. IEEE Access 7.

Anderson, H.S., Roth, P., 2018. EMBER: an Open Dataset for Training Static PE Malware Machine Learning Models. ArXiv e-prints.

Anderson, R., Barton, C., Böhme, R., Clayton, R., Ganán, C., Grasso, T., Levi, M., Moore, T., Vasek, M., 2019. Measuring the Changing Cost of Cybercrime.

Aryal, K., Gupta, M., Abdelsalam, M., 2021. A survey on adversarial attacks for malware analysis. ArXiv preprint arXiv:2111.08223.

Aryal, K., Gupta, M., Abdelsalam, M., 2022. Analysis of label-flip poisoning attack on machine learning based malware detector. In: IEEE International Conference on Big Data.

Bergstra, J., Bengio, Y., 2012. Random search for hyper-parameter optimization. J. Mach. Learn. Res. 13 (1).

Bergstra, J., Bardenet, R., Bengio, Y., Kégl, B., 2011. Algorithms for hyper-parameter optimization. Adv. Neural Inf. Process. Syst. 24.

Brown, P., Brown, A., Gupta, M., Abdelsalam, M., 2022. Online malware classification with system-wide system calls in cloud iaas. In: 2022 IEEE 23rd International Conference on Information Reuse and Integration for Data Science (IRI). IEEE, pp. 146–151.

Cheng, R., 2010. Random search in high dimensional stochastic optimization. In: Proceedings of the Winter Simulation Conference, Ser. WSC'10. Winter Simulation Conference.

Demme, J., et al., 2013. On the feasibility of online malware detection with performance counters. ACM SIGARCH Comput. Archit. News 41 (3).

Erickson, N., Mueller, J., Shirkov, A., Zhang, H., Larroy, P., Li, M., Smola, A., 2020. Autogluon-tabular: robust and accurate automl for structured data. ArXiv preprint arXiv:2003.06505.

Fan, Y., et al., 2016. Malicious sequential pattern mining for automatic malware detection. Expert Syst. Appl. 52.

Feurer, M., Hutter, F., 2018. Towards further automation in automl. In: ICML AutoML Workshop.

Ganesh, M., Pednekar, P., Prabhuswamy, P., Nair, D.S., Park, Y., Jeon, H., 2017. CNN-based Android malware detection. In: IEEE International Conference on Software Security and Assurance.

Guan, Q., Zhang, Z., Fu, S., 2012. Ensemble of Bayesian predictors and decision trees for proactive failure management in cloud computing systems. J. Commun.

Gupta, M., Akiri, C., Aryal, K., Parker, E., Praharaj, L., 2023. From chatgpt to threatgpt: impact of generative ai in cybersecurity and privacy. IEEE Access.

Harang, R., Rudd, E.M., 2020. Sorel-20m: a large scale benchmark dataset for malicious pe detection. ArXiv preprint arXiv:2012.07634.

He, K., Zhang, X., Ren, S., Sun, J., 2016. Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.

He, X., Zhao, K., Chu, X., 2021. AutoML: a survey of the state-of-the-art. Knowl.-Based Syst. 212.

Huang, W., Stokes, J.W., 2016. MtNet: a multi-task neural network for dynamic malware classification. In: Int. Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer.

Iandola, F., Moskewicz, M., Karayev, S., Girshick, R., Darrell, T., Keutzer, K., 2014. Densenet: implementing efficient convnet descriptor pyramids. ArXiv preprint arXiv:1404.1869.

Isingizwe, D.F., et al., 2021. Analyzing learning-based encrypted malware traffic classification with AutoML. In: IEEE Conference on Communication Technology.

Jha, S., et al., 2020. Recurrent neural network for detecting malware. Comput. Secur. 99.

Kamath, C.N., et al., 2018. Comparative study between traditional machine learning and deep learning approaches for text classification. In: ACM Symposium on Document Engineering.

Kimmel, J.C., Mcdole, A.D., Abdelsalam, M., Gupta, M., Sandhu, R., 2021. Recurrent neural networks based online behavioural malware detection techniques for cloud infrastructure. IEEE Access 9.

Kimmell, J.C., Abdelsalam, M., Gupta, M., 2021. Analyzing machine learning approaches for online malware detection in cloud. In: IEEE International Conference on Smart Computing.

Kolosnjaji, B., Zarras, A., Webster, G., Eckert, C., 2016. Deep learning for classification of malware system call sequences. In: Australasian Joint Conference on Artificial Intelligence. Springer.

Kundu, P.P., et al., 2021. An empirical evaluation of automated machine learning techniques for malware detection. In: ACM Workshop on Security and Privacy Analytics.

Li, L., Talwalkar, A., 2020. Random search and reproducibility for neural architecture search. In: Uncertainty in Artificial Intelligence. PMLR.

Liu, H., Simonyan, K., Yang, Y., 2018. Darts: differentiable architecture search. ArXiv preprint arXiv:1806.09055.

Loi, N., Borile, C., Ucci, D., 2021. Towards an automated pipeline for detecting and classifying malware through machine learning. ArXiv preprint arXiv:2106.05625.

Luckett, P., McDonald, J.T., Dawson, J., 2016. Neural network analysis of system call timing for rootkit detection. In: IEEE Cybersecurity Symposium.

McDole, A., Abdelsalam, M., Gupta, M., et al., 2020. Analyzing CNN based behavioural malware detection techniques on cloud IaaS. In: International Conference on Cloud Computing. Springer.

McDole, A., et al., 2021. Deep learning techniques for behavioral malware analysis in cloud IaaS. In: Malware Analysis Using Artificial Intelligence and Deep Learning. Springer.

Mishra, P., Khurana, K., Gupta, S., Sharma, M.K., 2019. Vmanalyzer: malware semantic analysis using integrated cnn and bi-directional lstm for detecting vm-level attacks in cloud. In: International Conference on Contemporary Computing (IC3).

Nath, H.V., Mehtre, B.M., 2014. Static malware analysis using machine learning methods. In: International Conference on Security in Computer Networks and Distributed Systems. Springer.

Nguyen, A.T., et al., 2021. Leveraging Uncertainty for Improved Static Malware Detection Under Extreme False Positive Constraints.

Ozsoy, M., Donovick, C., Gorelik, I., Abu-Ghazaleh, N., Ponomarev, D., 2015. Malware-aware processors: a framework for efficient online malware detection. In: 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA).
Pascanu, R., Stokes, J.W., Sanossian, H., Marinescu, M., Thomas, A., 2015. Malware classification with recurrent networks. In: IEEE Conference on Acoustics, Speech and Signal Processing.
Pham, H., Guan, M., Zoph, B., Le, Q., Dean, J., 2018. Efficient neural architecture search via parameters sharing. In: International Conference on Machine Learning. PMLR.
Raff, E., Sylvester, J., Nicholas, C., 2017. Learning the PE header, malware detection with minimal domain knowledge. In: ACM Workshop on Artificial Intelligence and Security.
Raff, E., et al., 2018a. Malware detection by eating a whole Exe. In: Workshops at AAAI Conference on Artificial Intelligence.
Raff, E., et al., 2018b. An investigation of byte N-gram features for malware classification. J. Comput. Virol. Hacking Tech. 14 (1).
Raff, E., et al., 2020. Classifying sequences of extreme length with constant memory applied to malware detection. ArXiv preprint arXiv:2012.09390.
Rezende, E., Ruppert, G., Carvalho, T., Theophilo, A., Ramos, F., Geus, P.d., 2018. Malicious software classification using vgg16 deep neural network's bottleneck features. In: Information Technology-New Generations. Springer.
Rudd, E.M., Ducau, F.N., Wild, C., Berlin, K., Harang, R., 2019. ALOHA: auxiliary loss optimization for hypothesis augmentation. In: USENIX Security Symposium.
Sahin, M., Bahtiyar, S., 2020. A survey on malware detection with deep learning. In: Int. Conf. on Security of Information and Networks.
Sewak, M., Sahay, S.K., Rathore, H., 2018. An investigation of a deep learning based malware detection system. In: International Conference on Availability, Reliability and Security.
Shalaginov, A., Banin, S., Dehghantanha, A., Franke, K., 2018. Machine learning aided static malware analysis: a survey and tutorial. In: Cyber Threat Intelligence. Springer, pp. 7–45.
Simonyan, K., Zisserman, A., 2014. Very deep convolutional networks for large-scale image recognition. ArXiv preprint arXiv:1409.1556.
Thomas, R., 2017. Lief - library to instrument executable formats. https://lief.quarkslab.com/.
Tobiyama, S., et al., 2016. Malware detection with deep neural network using process behavior. In: IEEE Computer Software and Applications Conference, vol. 2.
Vinayakumar, R., et al., 2019. Robust intelligent malware detection using deep learning. IEEE Access 7, 46 717–46 738.
Wang, W., Zhao, M., Wang, J., 2019. Effective Android malware detection with a hybrid model based on deep autoencoder and convolutional neural network. J. Ambient Intell. Humaniz. Comput. 10 (8).
Wei, T., Wang, C., Rui, Y., Chen, C.W., 2016. Network morphism. In: International Conference on Machine Learning.
Willems, C., et al., 2007. Toward automated dynamic malware analysis using Cwsandbox. IEEE Secur. Priv. 5 (2).
Xiao, X., Zhang, S., Mercaldo, F., Hu, G., Sangaiah, A.K., 2019. Android malware detection based on system call sequences and lstm. Multimed. Tools Appl.
Xie, W., Xu, S., Zou, S., Xi, J., 2020. A system-call behavior language system for malware detection using a sensitivity-based lstm model. In: Proceedings of the 2020 3rd International Conference on Computer Science and Software Engineering, pp. 112–118.
Xu, Z., et al., 2017. Malware detection using machine learning based analysis of virtual memory access patterns. In: IEEE Design, Automation & Test in Europe Conference & Exhibition.
Yeo, M., Koo, Y., Yoon, Y., Hwang, T., Ryu, J., Song, J., Park, C., 2018. Flow-based malware detection using convolutional neural network. In: IEEE Conference on Information Networking.

Austin Brown Received his B.S. in Computer Science from Tennessee Tech University in 2020. He received his M.S in Computer Science from Tennessee Tech University in 2022. His interests include deep learning, malware research, and cloud computing.

Maanak Gupta is an Assistant Professor in Computer Science at Tennessee Tech University, Cookeville, USA. He received M.S. and Ph.D. in Computer Science from the University of Texas at San Antonio (UTSA) and has also worked as a postdoctoral fellow at the Institute for Cyber Security (ICS) at UTSA. His primary area of research includes security and privacy in cyber space focused in studying foundational aspects of access control, malware analysis, AI and machine learning assisted cyber security, adversarial AI and their applications in technologies including cyber physical systems, cloud computing, IoT and Big Data. He holds a B.Tech degree in Computer Science and Engineering from Kuruskhetra University, India, and an M.S. in Information Systems from Northeastern University, Boston. He is senior member of IEEE.

Mahmoud Abdelsalam received the B.Sc. degree from the Arab Academy for Science and Technology and Maritime Transportation (AASTMT), in 2013, and the M.Sc. and Ph.D. degrees from the University of Texas at San Antonio (UTSA), in 2017 and 2018, respectively. He was working as a Postdoctoral Research Fellow with the Institute for Cyber Security (ICS), UTSA, and as an Assistant Professor with the Department of Computer Science, Manhattan College. He is currently working as an Assistant Professor with the Department of Computer Science, North Carolina A&T State University. His research interests include computer systems security, anomaly and malware detection, cloud computing security and monitoring, CPS security, and applied machine learning.