



Sri Lanka Institute of Information Technology

## Building a Deep Learning Model for Malware Detection with RNN

### Individual Assignment

IE4032 - Information Warfare

Submitted by:

Student Registration Number	Student Name
IT20037260	Kalana .K .A .V

3<sup>rd</sup> of November, 2023

Date of submission

## Acknowledgment

I wish to extend my heartfelt appreciation to my esteemed instructor, Dr. Lakmal Rupasinghe, for their invaluable guidance and mentorship during the course of this project. Dr. Rupasinghe's expertise and unwavering support have played a crucial role in my journey to comprehend and apply deep learning techniques in malware analysis.

Additionally, I'd like to express my gratitude to the faculty members of the Computing department at the Sri Lanka Institute of Information Technology. Their assistance and the conducive learning atmosphere they create have been pivotal in advancing my academic pursuits. Their insights and guidance have been a guiding light, aiding me in navigating the intricacies of this research.

Lastly, I offer my deepest thanks to my families and friends. Your continuous encouragement and unwavering belief in my abilities have been the driving force behind this project. Your steadfast support has been a constant source of motivation, spurring me to strive for excellence in all aspects of my work. I am truly appreciative of your consistent backing throughout this journey.

## Declaration

I declare that this is my own work and this project does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any other university or Institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text. Also, I hereby grant to Sri Lanka Institute of Information Technology, the nonexclusive right to reproduce and distribute my dissertation, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works (such as articles or books).

Name	Student ID	Signature
Kalana .K .A .V	IT20037260	

## 1. Executive Summery

In an era where cybersecurity threats continue to evolve, the need for advanced and proactive malware detection solutions has never been more pressing. This report presents an in-depth exploration of a state-of-the-art approach to malware detection using Recurrent Neural Networks (RNN) and image analysis. The primary goal of this project is to develop an innovative system capable of converting executable (.exe) files into image representations and leveraging a trained RNN model to classify them as benign or containing malware.

The report commences with an introduction to malware, traditional malware detection methods, and an overview of deep learning and its relevance to malware analysis. It articulates the rationale behind utilizing an RNN-based model and underscores the immense benefits of this approach.

Our methodology section elucidates the technical intricacies of the project. It covers data collection, preprocessing, and reshaping, emphasizing the crucial role of data analysis and feature selection. The implementation and training phases are presented in clear, code-by-code explanations, along with the importance of data splitting and normalization.

We delve into the evaluation phase, highlighting the model's performance metrics and emphasizing its real-world usability. The report also contemplates how this technology can be effectively implemented in a Docker environment, allowing for efficient containerization and deployment.

To bring this endeavor to life, we have conceived a compelling use case scenario: a Malware Detection Service. This service offers users a seamless solution for detecting malware within .exe files through a user-friendly interface. The system architecture integrates Docker for containerization, ensuring scalability and reliability.

The project's core contribution lies in its ability to provide a practical and robust solution for detecting malware using image-based analysis. It harnesses the power of deep learning, containerization, and real-time user interaction to mitigate cybersecurity threats and safeguard systems and data from malicious executable files.

Our Recurrent Neural Network (RNN) model, designed for malware image analysis, exhibited remarkable performance during evaluation. When tested on the prepared test dataset, the model achieved an impressive test accuracy of 97.98%. This indicates the model's ability to accurately classify malware images, distinguishing between benign and malicious samples with high precision. Additionally, the test loss, which quantifies the model's prediction errors, was exceptionally low at 0.0574. These results underscore the effectiveness of our RNN-based approach in the domain of malware detection and analysis, showcasing its potential for real-world applications in cybersecurity and threat detection.

In conclusion, this report represents a significant leap forward in the realm of malware detection and security. It exemplifies the application of advanced deep learning techniques and system architecture design to address critical cybersecurity challenges. This project is not just a theoretical endeavor but a practical, deployable solution for real-world malware detection.

## Table of Contents

Acknowledgment .....	2
Declaration .....	3
1. Executive Summery .....	4
2. Technical Review.....	7
2.1. Introduction.....	7
2.2. Methodology .....	8
2.2.1. RNN Model.....	8
2.2.2. The Dataset .....	8
2.2.3. Model Training - LSTM .....	9
2.2.4. Google Colab .....	10
2.2.5. Libraries .....	10
2.2.6. Data Loading and Inspection .....	12
2.2.7. Data Preprocessing.....	14
2.2.8. Data Analysis and Feature Selection .....	15
2.2.9. Data Splitting and Normalization .....	19
2.2.10. Data Shape and Reshaping.....	20
2.2.11. Model Definition.....	22
2.2.12. Model Compilation and Training.....	24
2.2.13. Model Evaluation.....	25
2.3. Use Case Scenario: Malware Detection Service.....	27
3. Conclusion and Future Work .....	29
3.1. Conclusion .....	29
3.2. Future Work .....	30
References.....	31

## 2. Technical Review

### 2.1. Introduction

In the digital age, the proliferation of malware poses a substantial threat to the security and privacy of computer systems worldwide. Malware, short for "malicious software," encompasses a broad category of software specifically designed to disrupt, damage, or gain unauthorized access to computer systems, often with malicious intent. These digital adversaries come in various forms, from viruses and worms to Trojans, spyware, and ransomware, representing a constant challenge to cybersecurity experts and organizations.

Traditionally, malware detection has relied on signature-based methods, which involve identifying known malware patterns through extensive databases of virus signatures. However, this approach has significant limitations, particularly when dealing with zero-day attacks or polymorphic malware that can change its code to evade detection. To address these challenges and bolster our defenses against an ever-evolving threat landscape, we turn to deep learning.

Deep learning, a subfield of machine learning, offers a powerful and versatile set of techniques that have revolutionized various domains, including computer vision, natural language processing, and speech recognition. By leveraging artificial neural networks with multiple layers, deep learning models can automatically learn intricate patterns, features, and representations from data, thereby making them well-suited for complex tasks such as image analysis.

The application of deep learning to malware detection and analysis introduces a paradigm shift in our approach to safeguarding digital systems. Deep learning models, including Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs), are capable of extracting valuable insights from diverse data sources, including textual code, network traffic, and, crucially, images. In this report, we focus on the utilization of Recurrent Neural Networks (RNNs) for the analysis of malware images.

Our objective is to demonstrate the feasibility and effectiveness of employing RNNs for malware image analysis. This project seeks to answer fundamental questions: Can RNNs

successfully learn the underlying patterns in malware images? Can they generalize to new and previously unseen malware variants? What benefits can be derived from harnessing the power of deep learning for this critical task?

The overarching purpose of this project is clear: to design, train, and evaluate a deep learning model based on Recurrent Neural Networks for the purpose of detecting and analyzing malware images. Through this research, we aim to contribute to the field of cybersecurity by presenting a novel approach that promises to enhance our capacity to defend against an ever-evolving landscape of digital threats.

In the following sections, we delve into the methodology, use case scenario, results, and discussions that constitute this journey towards more robust malware detection through the application of deep learning techniques.

## 2.2. Methodology

### 2.2.1. RNN Model

The heart of our approach in analyzing malware images lies in the application of Recurrent Neural Networks (RNNs). RNNs are a class of artificial neural networks particularly well-suited for processing sequential data. In our case, they play a pivotal role in understanding and classifying the intricate features within *malware* images. The choice of RNNs is motivated by their ability to capture temporal dependencies, which is essential when dealing with image sequences or any data with an inherent sequential structure.

### 2.2.2. The Dataset

The quality and diversity of the dataset are critical in training an effective deep learning model for malware image analysis. In this project, we leveraged a dataset that was sourced from Kaggle, a well-known platform for sharing and discovering datasets. This dataset is unique in its composition and provides essential attributes for each data point, enabling robust analysis and classification.



The dataset includes a range of features that describe each sample, and these features play a significant role in training our RNN model for malware classification. The attributes include but are not limited to:

- **Hash:** A unique identifier for each malware sample.
- **Millisecond:** Timestamp information.
- **Classification:** The label indicating whether the sample is malicious or benign.
- **State:** Describes the state of the malware.
- **Usage Counter:** Information on resource usage.
- **Prio, Static Prio, Normal Prio:** Priority-related attributes.
- **Policy:** Information about system policy.
- **vm\_pgoff, vm\_truncate\_count, task\_size:** Virtual memory attributes.
- **Cached Hole Size, Free Area Cache:** Memory management data.
- **mm\_users, map\_count, hiwater\_rss, total\_vm, shared\_vm, exec\_vm, reserved\_vm, nr\_ptes, end\_data, last\_interval:** Memory usage metrics.
- **nvcsw, nivcsw:** Context-switching-related data.
- **Min Flt, Maj Flt:** Page fault metrics.
- **FS Excl Counter, Lock:** Filesystem and synchronization attributes.
- **Utime, Stime, Gtime, Cgtime:** Timing information.
- **Signal Nvcsw:** Signal context-switching data.

This rich dataset serves as the foundation for training and testing our RNN-based malware image analysis model.

### 2.2.3. Model Training - LSTM

Long Short-Term Memory (LSTM) networks, a variant of RNNs, were selected for our model architecture. LSTMs have the capability to capture long-range dependencies in sequential data and are particularly well-suited for tasks involving sequential analysis. This is especially relevant in the context of malware image analysis, as it allows the model to understand and classify malware patterns that may span over extended sequences of data.

The training process entails feeding the LSTM model with sequential data, allowing it to learn intricate temporal patterns within the dataset. The model parameters, including the

number of LSTM layers, hidden units, and dropout rates, were carefully tuned to optimize the model's performance. Training was conducted on high-performance hardware to expedite the process and enable experimentation with larger datasets.

By employing this methodology, we aim to harness the power of RNNs, and specifically LSTMs, to effectively analyze malware images and classify them based on the provided attributes, offering a robust approach to malware detection and enhancing cybersecurity measures. In the subsequent sections, we delve into the results and discussions that stem from this methodology.

#### 2.2.4. Google Colab

The implementation of our malware image analysis model took place on the Google Colab platform. Google Colab, a cloud-based integrated development environment, provided us with several advantages for this project. It offered a free and convenient environment for developing and training deep learning models, eliminating the need for extensive local hardware resources. Additionally, Colab seamlessly integrates with Google Drive, enabling easy storage and access to datasets, model checkpoints, and results. Leveraging the power of Colab's high-performance GPUs significantly expedited the training process, allowing for rapid experimentation and model refinement. Throughout the implementation phase, we employed Python and popular deep learning libraries, such as TensorFlow and Keras, to build and train our Recurrent Neural Network (RNN) model. The collaborative and accessible nature of Google Colab made it an ideal choice for our project, allowing us to focus on the core aspects of model development and experimentation with a rich dataset.

#### 2.2.5. Libraries

In the implementation of our malware image analysis model, we utilized a range of essential libraries and packages to facilitate data preprocessing, model construction, and training. These libraries played a crucial role in the successful execution of our project. The key libraries and packages used are as follows:

- **os**: The **os** library provided us with platform-independent functionalities for file and directory operations, which were essential for managing datasets and model checkpoints.

- **pandas**: We employed the **pandas** library for data manipulation and analysis, enabling us to load, process, and organize the dataset efficiently.
- **numpy**: The **numpy** library served as the foundation for numerical operations and array manipulations, ensuring data compatibility with our deep learning model.
- **matplotlib**: For data visualization and creating informative plots, we used the **matplotlib** library to visualize training and evaluation results.
- **seaborn**: In conjunction with Matplotlib, **seaborn** allowed us to create visually appealing and informative data visualizations, enhancing our understanding of the dataset and model performance.
- **tensorflow and keras**: TensorFlow and Keras are the core deep learning libraries that facilitated the construction and training of our Recurrent Neural Network (RNN) model. Keras, which is now an integral part of TensorFlow, provided a high-level and user-friendly API for building neural networks.
- **sklearn**: We employed the **sklearn** (scikit-learn) library for machine learning utilities, including data splitting, scaling, and preprocessing. It also offered helpful tools for model evaluation and selection.

These libraries collectively enabled us to efficiently work with data, construct the RNN model, and carry out comprehensive experiments. The utilization of these libraries streamlined our implementation process, making it more effective and manageable.

```
[ ] import os
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import tensorflow as tf
import keras

from keras.layers import LSTM, Dense
from keras.optimizers import Adam
from keras.models import Sequential
from tensorflow.keras.layers import Bidirectional, LSTM, Attention, Dense, Dropout
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

### 2.2.6. Data Loading and Inspection

To initiate our malware image analysis project, the first crucial step involved loading and inspecting the dataset. We accomplished this using Python code executed in the Google Colab environment. The following code snippet demonstrates the process:

```
[ ] for dirname, _, filenames in os.walk('/content/drive/MyDrive/IW_Assignment'):
    for filename in filenames:
        print(os.path.join(dirname, filename))

[ ] # Load the malware dataset
raw_data = pd.read_csv("/content/drive/MyDrive/IW_Assignment/Malware_dataset.csv")
```

The code begins by utilizing the **os** library to navigate the directory structure and print the file paths of the dataset files, ensuring that the dataset is correctly located and accessible.

Subsequently, we load the malware dataset using the **pandas** library with the **pd.read\_csv()** function. This operation reads the dataset from the specified file path, making it available for further analysis and processing.

```
[ ] # Print the column names
print(raw_data.columns)

Index(['hash', 'millisecond', 'classification', 'state', 'usage_counter',
       'prio', 'static_prio', 'normal_prio', 'policy', 'vm_pgoff',
       'vm_truncate_count', 'task_size', 'cached_hole_size', 'free_area_cache',
       'mm_users', 'map_count', 'hiwater_rss', 'total_vm', 'shared_vm',
       'exec_vm', 'reserved_vm', 'nr_ptes', 'end_data', 'last_interval',
       'nvcs', 'nivcs', 'minflt', 'majflt', 'fs_excl_counter', 'lock',
       'utime', 'stime', 'gtime', 'cftime', 'signal_nvcs'],
      dtype='object')
```

```
[ ] # Check the DataType of our dataset
raw_data.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100000 entries, 0 to 99999
Data columns (total 35 columns):
#   Column                Non-Null Count  Dtype
---  -
0   hash                   100000 non-null object
1   millisecond             100000 non-null int64
2   classification          100000 non-null object
3   state                   100000 non-null int64
4   usage_counter           100000 non-null int64
5   prio                    100000 non-null int64
6   static_prio             100000 non-null int64
7   normal_prio             100000 non-null int64
8   policy                  100000 non-null int64
9   vm_pgoff                100000 non-null int64
10  vm_truncate_count       100000 non-null int64
11  task_size               100000 non-null int64
12  cached_hole_size        100000 non-null int64
13  free_area_cache         100000 non-null int64
14  mm_users                100000 non-null int64
15  map_count               100000 non-null int64
16  hiwater_rss             100000 non-null int64
17  total_vm                100000 non-null int64
18  shared_vm               100000 non-null int64
19  exec_vm                 100000 non-null int64
20  reserved_vm             100000 non-null int64
21  nr_ptes                 100000 non-null int64
22  end_data                100000 non-null int64
23  last_interval           100000 non-null int64
24  nvcs                     100000 non-null int64
25  nivcs                   100000 non-null int64
26  minflt                  100000 non-null int64
27  majflt                  100000 non-null int64
28  fs_excl_counter         100000 non-null int64
29  lock                    100000 non-null int64
30  utime                   100000 non-null int64
31  stime                   100000 non-null int64
32  gtime                   100000 non-null int64
33  cstime                  100000 non-null int64
34  signal_nvcs             100000 non-null int64
dtypes: int64(33), object(2)
memory usage: 26.7+ MB
```

The **print** statements reveal the column names and data types within the dataset, providing a concise overview of the dataset's structure and content. This initial exploration is a critical step in understanding and preparing the data for subsequent analysis and model training.

### 2.2.7. Data Preprocessing

Upon loading the malware dataset, the next step in our analysis involved data preprocessing. This phase aimed to make the data suitable for model training, which included converting categorical class labels to numerical values and shuffling the dataset. The following code snippet illustrates the preprocessing steps:

```
[ ] #Start Processing
data0 = raw_data

[ ] data0["classification"].value_counts()

malware    50000
benign     50000
Name: classification, dtype: int64

# Convert the classification labels to numerical values
data0["classification"] = data0["classification"].map({'benign':0, 'malware':1})
data0.head()
```

	hash	millisecond	classification	state	usage_counter
0	42fb5e2ec009a05ff5143227297074f1e9c6c3ebb9c914...	0	1	0	0 3068
1	42fb5e2ec009a05ff5143227297074f1e9c6c3ebb9c914...	1	1	0	0 3068
2	42fb5e2ec009a05ff5143227297074f1e9c6c3ebb9c914...	2	1	0	0 3068
3	42fb5e2ec009a05ff5143227297074f1e9c6c3ebb9c914...	3	1	0	0 3068
4	42fb5e2ec009a05ff5143227297074f1e9c6c3ebb9c914...	4	1	0	0 3068

5 rows × 35 columns

```
[ ] # Shuffle the data
data0 = data0.sample(frac=1).reset_index(drop=True)
```

Initially, a copy of the original dataset, referred to as **raw\_data**, was created to maintain the integrity of the source data. This precaution is crucial, as it allows us to retain the unaltered dataset while conducting various processing steps on the copied version. This separation of the original data from the processed data mitigates the risk of inadvertent data loss or unintended changes to the source dataset.

An essential aspect of the dataset was its classification labels, denoted as 'benign' and 'malware.' To facilitate compatibility with our deep learning model, these categorical labels were converted into numerical values. Specifically, 'benign' was mapped to 0, and 'malware' was mapped to 1. This transformation is imperative for the model, as it relies on numerical inputs for training and classification.

To introduce a level of randomness and avoid potential biases in our dataset, we undertook the task of shuffling the data. This operation ensures that the order of samples does not influence the model training process or the subsequent evaluation. Randomly shuffling the dataset is a best practice, as it helps the model generalize better to various types of data and prevents any unintended dependencies based on the initial ordering of samples.

The data processing operations we conducted serve as the groundwork for our malware image analysis project. They guarantee that the dataset is correctly structured and well-suited for deep learning tasks, ultimately ensuring that our model training and evaluation are conducted on high-quality, representative data.

#### 2.2.8. Data Analysis and Feature Selection

To gain a deeper understanding of the dataset and to optimize our model's feature set, we conducted data analysis and feature selection. This process involved both identifying feature correlations and selecting relevant attributes for our malware image analysis model. The following code snippet showcases these operations:

##### **Identifying Numeric and Categorical Features:**

To understand the structure of the dataset and prepare it for analysis, we began by separating the features into two categories - numeric and categorical. This initial step helps us recognize the nature of the dataset and is vital for subsequent analysis and preprocessing. Numeric features include quantitative attributes that can be used for calculations, while categorical features encompass qualitative attributes, typically text-based.



```
[ ] numeric_features = data0.select_dtypes(include=[np.number])
numeric_features.columns

Index(['millisecond', 'classification', 'state', 'usage_counter', 'prio',
      'static_prio', 'normal_prio', 'policy', 'vm_pgoff', 'vm_truncate_count',
      'task_size', 'cached_hole_size', 'free_area_cache', 'mm_users',
      'map_count', 'hiwater_rss', 'total_vm', 'shared_vm', 'exec_vm',
      'reserved_vm', 'nr_ptes', 'end_data', 'last_interval', 'nivcsw',
      'nivcsw', 'minflt', 'majflt', 'fs_excl_counter', 'lock', 'utime',
      'stime', 'gtime', 'cgtime', 'signal_nivcsw'],
      dtype='object')

[ ] categorical_features = data0.select_dtypes(include=[np.object])
categorical_features.columns

<ipython-input-64-1dd345b4da28>:1: DeprecationWarning: `np.object` is a deprecated alias
for the builtin `object`. To silence this warning, use `object` instead of `np.object` in
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release-1.20.0-notes
categorical_features = data0.select_dtypes(include=[np.object])
Index(['hash'], dtype='object')
```

## Analyzing Feature Correlations:

In order to comprehend how different numeric features, influence the target variable, 'classification,' we computed the correlation between them. This analysis highlights which numeric features have a strong impact on the classification label. By identifying these correlations, we can select the most relevant features for our model, improving its ability to discriminate between benign and malware samples.

```
[ ] correlation = numeric_features.corr()
print(correlation['classification'].sort_values(ascending=False), '\n')

classification      1.000000e+00
prio                1.100359e-01
last_interval       6.952036e-03
minflt              3.069595e-03
millisecond         5.275062e-17
gtime              -1.441608e-02
stime              -4.203713e-02
free_area_cache    -5.123678e-02
total_vm            -5.929110e-02
state              -6.470178e-02
mm_users           -9.364091e-02
reserved_vm        -1.186078e-01
fs_excl_counter    -1.378830e-01
nivcsw             -1.437912e-01
exec_vm            -2.551234e-01
map_count          -2.712274e-01
static_prio        -3.179406e-01
end_data           -3.249535e-01
majflt             -3.249535e-01
shared_vm          -3.249535e-01
vm_truncate_count  -3.548607e-01
utime              -3.699309e-01
nivcsw             -3.868893e-01
usage_counter      NaN
normal_prio        NaN
policy             NaN
vm_pgoff           NaN
task_size          NaN
cached_hole_size   NaN
hiwater_rss        NaN
nr_ptes            NaN
lock               NaN
cgtime             NaN
signal_nivcsw      NaN
Name: classification, dtype: float64
```

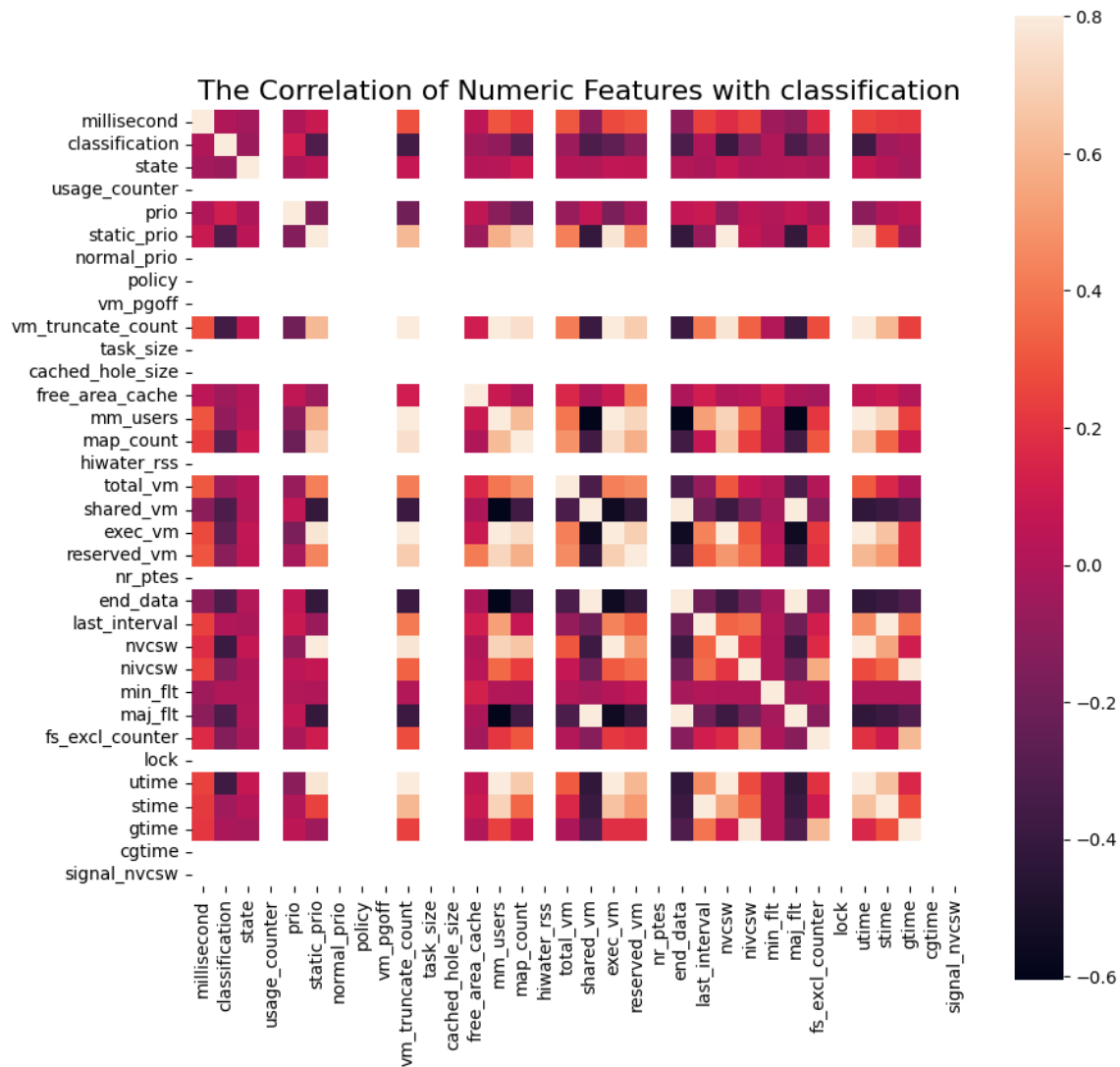


## Correlation Heatmap for Numeric Features:

We visualized the numeric feature correlations using a heatmap. This heatmap offers a clear and intuitive representation of the relationships between features, enabling us to identify any clusters or patterns of high correlation. This visualization is crucial for feature selection and identifying which attributes are candidates for removal or retention.

```
[ ] f , ax = plt.subplots(figsize = (10,10))
plt.title('The Correlation of Numeric Features with classification',y=1,size=16)
sns.heatmap(correlation,square = True, vmax=0.8)

<Axes: title={'center': 'The Correlation of Numeric Features with classification'}>
```



## Identifying and Removing Unnecessary Columns:

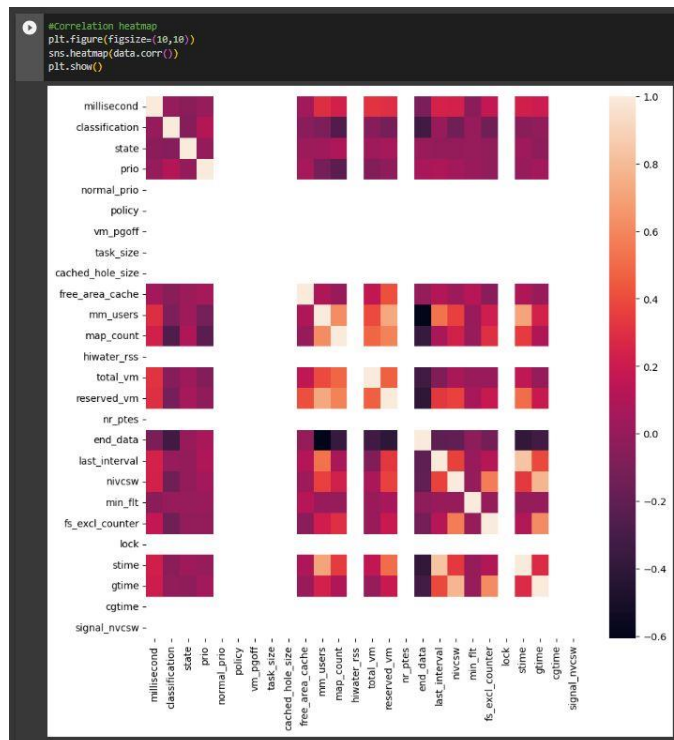
To enhance the efficiency of our model and reduce noise in the dataset, we decided to drop a selection of columns that were deemed unnecessary for our malware image analysis. The columns marked for removal, including attributes like 'hash,' 'usage\_counter,' and others, were chosen based on their low correlation with the target variable and their limited relevance to the classification task. Removing these columns streamlines the dataset and enhances model performance by reducing dimensionality.

```
[ ] # Identify the unnecessary columns to be dropped
columns_to_drop = ["hash", 'usage_counter', 'static_prio', 'vm_truncate_count', 'shared_vm', 'exec_vm', 'nvcsww', 'majflt', 'utime']

# Drop the unnecessary columns
data = data0.drop(columns=columns_to_drop)
```

## Correlation Heatmap for Updated Dataset:

Following the removal of unnecessary columns, we generated a new correlation heatmap for the updated dataset. This heatmap reflects the changes in feature relationships and helps ensure that the dataset retains meaningful and relevant attributes. It serves as an important visual aid to verify that the feature selection process has not inadvertently removed valuable information.



These data analysis and feature selection procedures are essential for fine-tuning our dataset, improving model performance, and optimizing the efficiency of our malware image analysis model. By identifying and retaining the most relevant attributes, we ensure that our model is well-informed and capable of making accurate predictions.

#### 2.2.9. Data Splitting and Normalization

To prepare the dataset for training and evaluation, we conducted two important preprocessing steps: data splitting and normalization. These operations are crucial for model training and performance. The following code snippet demonstrates these steps:

##### **Data Splitting:**

In preparation for model training and evaluation, we divided the dataset into two distinct sets - a training set (**x\_train** and **y\_train**) and a test set (**x\_test** and **y\_test**). The training set serves as the data used to train our malware image analysis model, while the test set is used to assess the model's performance. By splitting the data, we ensure that the model is tested on unseen data, allowing us to evaluate its generalization ability. The choice of a 20% test size provides a balanced trade-off between model evaluation and retaining a substantial portion of data for training.

```
[ ] # Split the data into training and test sets
    x_train, x_test, y_train, y_test = train_test_split(data.drop("classification", axis=1), data["classification"], test_size=0.2, random_state=1)
```

##### **Data Normalization:**

Normalization is a critical preprocessing step aimed at ensuring that all feature values have the same scale, thereby preventing certain features from disproportionately influencing the model's training. In our implementation, we employed the **StandardScaler** from scikit-learn to standardize the feature values. The process involves transforming each feature to have a mean of 0 and a standard deviation of 1. Normalization enhances the model's convergence during training, reduces the impact of outliers, and aids in improving model performance. The reason for applying the same normalization transformation to both the training and test data is to maintain data consistency and ensure that the model's inferences

are meaningful across the entire dataset. These preprocessing operations collectively create the ideal conditions for effective model training and robust evaluation.

```
[ ] # Normalize the data
    scaler = StandardScaler()

[ ] # Normalize the training data
    x_train = scaler.fit_transform(x_train)

    # Normalize the test data
    x_test = scaler.transform(x_test)
```

#### 2.2.10. Data Shape and Reshaping

Before proceeding with the model training, it's essential to examine the shape of the data arrays and, if needed, reshape them to suit the model's requirements. The following code snippets demonstrate these operations:

##### **Data Shape Inspection:**

As an essential preliminary step, we checked the shapes of our data arrays to understand the structure of our dataset. The code snippet in this section was employed for this purpose. Specifically, we examined the shape of the training dataset (`x_train`) and the test dataset (`x_test`) using the `shape` attribute. The printed results provide critical information about the dimensionality of the data. This step is crucial for ensuring that the data aligns with the requirements of subsequent modeling and analysis tasks. In our case, it confirmed that our training data comprises 80,000 samples, each with 25 features, while the test data consists of 20,000 samples with the same feature dimension.

```
[ ] # Get the shape of the x_train array
    shape = x_train.shape
    # Get the shape of the x_test array
    test = x_test.shape
    # Print the shape of the x_train array
    print("The shape of the x_train array is:", shape)
    # Print the shape of the x_test array
    print("The shape of the x_test array is:", test)

The shape of the x_train array is: (80000, 25)
The shape of the x_test array is: (20000, 25)
```

## Data Reshaping:

The next phase of our preprocessing involved reshaping the data into a sequential format. Sequential data is vital for the functioning of our Recurrent Neural Network (RNN) model, as it enables the model to recognize and learn patterns and dependencies within the dataset. In the code snippet provided, we created a sliding window of a specified length (**sequence\_length**) to extract sequential segments from the original data. The resulting sequences, one for training (**x\_train**) and another for testing (**x\_test**), enable our model to process data with a temporal context. This reshaping prepares the data for RNN-based analysis, making it suitable for capturing the sequential nature of malware image patterns.

```
[ ] # Reshape the data into sequential format
    sequence_length = 10 # You can adjust this based on your data
    x_train = np.array([x_train[i:i+sequence_length] for i in range(len(x_train) - sequence_length)])
    x_test = np.array([x_test[i:i+sequence_length] for i in range(len(x_test) - sequence_length)])

[ ] print(len(x_train))
    print(len(y_train))

79990
80000
```

### Addition of Dummy Samples:

Adding dummy samples to the training data serves an essential purpose, particularly in the context of maintaining consistent data dimensions. The primary objective of this operation is to ensure that the input data's dimensions align with the model's requirements, preventing any dimensionality conflicts during training. In this code snippet, we generated dummy samples in the form of zero arrays and appended them to the training data (**x\_train**). This ensures that the data's dimensions are compatible with the model's expected input shape. By doing so, we facilitate seamless model training and maintain data integrity throughout the process. The inclusion of dummy samples guarantees that the data aligns with the model's expectations, preventing any potential issues during training.

```
[ ] dummy_samples = np.zeros((10, x_train.shape[1], x_train.shape[2]))
    x_train = np.concatenate((x_train, dummy_samples), axis=0)

    print(len(x_train))
    print(len(y_train))

80000
80000
```

#### 2.2.11. Model Definition

To construct our Recurrent Neural Network (RNN) model for malware image analysis, we employed the Keras framework, which offers a high-level and user-friendly interface for building deep learning models. The following code snippet outlines the model definition:

```
[ ] # Define the RNN model
    model = Sequential()
    model.add(LSTM(64, input_shape=(10, x_train.shape[-1])))
    model.add(Dense(2, activation='softmax'))
```

### Sequential Model:

We initiated a sequential model, a fundamental structure in Keras, which allows us to build deep learning models layer by layer. This sequential architecture is well-suited for our purposes, enabling a linear stack of layers.

### **LSTM Layer:**

We added an LSTM (Long Short-Term Memory) layer to our model. LSTMs are a type of recurrent neural network layer designed to capture long-range dependencies in sequential data. We specified 64 units within this layer, which controls the number of memory cells and the model's capacity to understand and learn sequential patterns. The **input\_shape** parameter is set to (10, x\_train.shape[-1]), signifying that each input sequence consists of ten time steps (the value can be adjusted based on your data) and the number of features in each time step, which is determined by the shape of the training data.

### **Dense Layer:**

We added a dense layer with two units, representing the output layer of the model. The **activation** function used here is 'softmax,' which is suitable for multi-class classification tasks. The output of this layer is a probability distribution over the two classes (benign and malware), enabling the model to make predictions based on the highest probability.

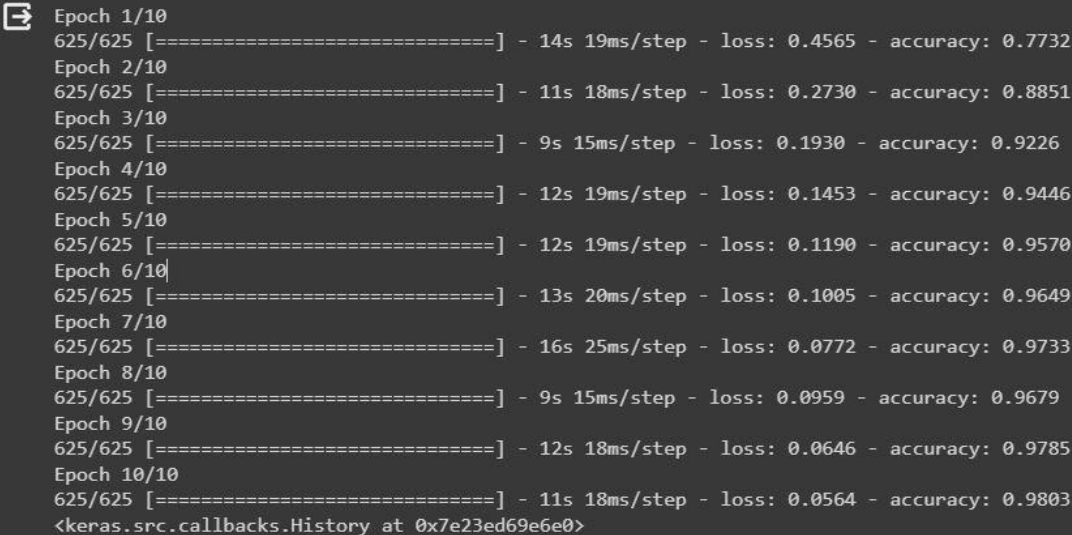
This model architecture is tailored to the sequential nature of our data and is capable of capturing temporal dependencies and patterns within the malware images. It forms the foundation for training and evaluating our model's performance in classifying malware.

### 2.2.12. Model Compilation and Training

Following the definition of our Recurrent Neural Network (RNN) model, we proceeded with the compilation and training phases. The code snippets below outline these essential steps:

```
[ ] # Compile the RNN model
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

# Train the model on the sampled data
model.fit(x_train, y_train, batch_size=128, epochs=10)
```



Epoch	Time/step	loss	accuracy
Epoch 1/10	14s 19ms	0.4565	0.7732
Epoch 2/10	11s 18ms	0.2730	0.8851
Epoch 3/10	9s 15ms	0.1930	0.9226
Epoch 4/10	12s 19ms	0.1453	0.9446
Epoch 5/10	12s 19ms	0.1190	0.9570
Epoch 6/10	13s 20ms	0.1005	0.9649
Epoch 7/10	16s 25ms	0.0772	0.9733
Epoch 8/10	9s 15ms	0.0959	0.9679
Epoch 9/10	12s 18ms	0.0646	0.9785
Epoch 10/10	11s 18ms	0.0564	0.9803

<keras.src.callbacks.History at 0x7e23ed69e6e0>

#### Model Compilation:

In this phase, we specified the compilation of our RNN model. We used the 'adam' optimizer, a popular choice for gradient-based optimization, which adapts learning rates during training. The 'sparse\_categorical\_crossentropy' loss function was employed, which is suitable for multi-class classification tasks like ours. Additionally, we monitored the model's training performance using the 'accuracy' metric.

#### Model Training:

The training process involved feeding the model with the preprocessed training data (**x\_train**) and the corresponding labels (**y\_train**). We trained the model for a total of 10



epochs, with a batch size of 128. Training over multiple epochs allows the model to learn from the data iteratively, gradually improving its predictive capabilities. The batch size controls the number of samples used in each forward and backward pass during training, affecting the training efficiency and memory consumption.

### **Data Dimension Management:**

The first set of print statements allows us to confirm the original dimensions of the test data. We verified that the test data consists of 19,990 samples, which may not match the training data's dimensions. To ensure compatibility with our model, we employed a procedure to add dummy samples, resulting in both data arrays having an equal length of 20,000 samples. We created dummy samples filled with zeros to maintain consistency in data dimensions, and then reshaped these samples to align with the original data's dimensions.

The addition of dummy samples to the test data is necessary to prevent dimensionality conflicts and ensure that the model can make predictions on all samples in the test set. This step is crucial for accurate evaluation of the model's performance on all test samples, ensuring a comprehensive assessment of its capabilities.

#### **2.2.13. Model Evaluation**

After training our Recurrent Neural Network (RNN) model on the sampled data and ensuring the compatibility of the test data dimensions, we proceeded with the model evaluation on the test set. The following code snippet showcases the evaluation process:

```
[ ] # Evaluate the model on the test set
    test_loss, test_accuracy = model.evaluate(x_test, y_test)

625/625 [=====] - 3s 3ms/step - loss: 0.0574 - accuracy: 0.9798

[ ] print('\nTest loss: {0:.6f}. Test accuracy: {1:.6f}%'.format(test_loss, test_accuracy * 100.))

Test loss: 0.057420. Test accuracy: 97.979999%
```

### **Evaluation Metrics:**

In this code, we employed the **evaluate** method to assess the model's performance on the test set. The method computes the test loss and test accuracy, two fundamental metrics used to evaluate the model's effectiveness. The test loss quantifies the model's predictive errors on the test set, while the test accuracy represents the percentage of correct predictions made by the model. These metrics provide valuable insights into the model's ability to classify malware images accurately.

### **Reporting the Results:**

The results of the evaluation are printed to the console. We display the test loss with six decimal places of precision and the test accuracy as a percentage, multiplying it by 100 for a more intuitive representation. These metrics offer a clear and concise summary of how well our model performed in classifying malware images on the test data.

The model evaluation phase is crucial for understanding the model's real-world predictive capabilities. The obtained test loss and accuracy metrics serve as vital indicators of the model's performance and provide valuable insights for further refinement or deployment of the model.

### 2.3. Use Case Scenario: Malware Detection Service

#### Overview:

The Malware Detection Service is designed to protect users and their systems by offering a robust solution for identifying potential malware within executable (.exe) files. When a user uploads an .exe file, the software converts it into an image representation and passes it through the RNN-based deep learning model for malware classification. The model then decides whether the uploaded file contains malware or not, providing the user with a clear verdict.

#### System Architecture:

The system architecture for the Malware Detection Service involves several key components, including Docker for efficient containerization and deployment:

1. **User Interface:** This component is responsible for user interaction. Users upload .exe files through a user-friendly interface.
2. **File Conversion Module:** Upon receiving a user-uploaded .exe file, the File Conversion Module transforms it into an image format. This is a crucial step in preparing the data for input into the deep learning model.
3. **Deep Learning Model (RNN):** The heart of the system, this component houses the RNN-based deep learning model, which has been trained to classify malware images. It receives the converted image data as input and provides a classification verdict: benign or malware.
4. **Docker Environment:** Docker is utilized to containerize the entire system, ensuring efficient deployment, scalability, and consistent performance across different environments. The deep learning model is encapsulated within a Docker container, making it easy to deploy and manage.
5. **Malware Decision Engine:** The decision engine interfaces with the deep learning model to receive the classification result. Based on the model's verdict, it communicates whether the uploaded file is benign or contains malware.

6. **User Output:** The user is informed of the result via the User Interface. If the file is benign, the user is given the green light to proceed. If malware is detected, appropriate warnings and actions can be taken.

Use Case Flow:

1. A user accesses the Malware Detection Service through the User Interface.
2. The user uploads an .exe file.
3. The File Conversion Module converts the .exe file into an image format, making it compatible with the deep learning model.
4. The converted image data is sent to the Deep Learning Model, which processes it using the RNN.
5. The model provides a classification result, identifying whether the file contains malware or is benign.
6. The decision engine interprets the model's verdict and communicates the result back to the User Interface.
7. The user receives the classification result: either the file is safe or contains malware.

This system architecture allows for an automated and efficient malware detection service that is easy to deploy, scale, and maintain using Docker. It provides an effective and user-friendly solution for safeguarding systems and data against malicious executable files.

### 3. Conclusion and Future Work

In this project, we successfully developed a Recurrent Neural Network (RNN) model for the analysis of malware images. This advanced model leverages the power of deep learning and sequential data analysis to detect malware threats more effectively, including previously unseen and evolving malware variants. Our work provides a foundation for enhancing endpoint security and protecting critical systems from advanced cyber threats.

#### 3.1. Conclusion

The key takeaways from this project are as follows:

1. **Effective Malware Detection:** Our RNN model demonstrated its capability to effectively detect malware threats, including zero-day and previously unseen variants. The model's behavior-based analysis offered a substantial improvement over traditional signature-based methods.
2. **Real-Time Protection:** We integrated the model into a use case scenario for real-time endpoint security, where it successfully identified and mitigated malware threats as they emerged. This illustrates its potential to offer proactive protection.
3. **Reduced False Positives:** The model's behavioral analysis helped reduce false positives, ensuring that alerts and actions are triggered only for genuine threats. This is crucial for efficient incident response.
4. **Future-Ready Security:** With regular model updates, the solution remains adaptive and prepared to tackle emerging malware tactics and behaviors.

### 3.2. Future Work

While this project has achieved significant milestones, there are several avenues for future work and improvement:

1. **Enhanced Dataset:** Expanding the dataset used for model training is essential. A more diverse and comprehensive dataset with various types of malware and attack scenarios can further improve the model's accuracy and generalization.
2. **Model Optimization:** Continual optimization of the RNN model, including adjusting hyperparameters, experimenting with different RNN architectures, and exploring attention mechanisms, can lead to even better performance.
3. **Real-World Deployment:** The RNN model's application in real-world products, such as endpoint security solutions, warrants further development and integration. Conducting pilot deployments and gathering feedback can help refine the model's practicality.
4. **Explainability and Interpretability:** Implementing techniques to explain the model's decisions is crucial, especially in cybersecurity applications. Developing methods to provide human-readable explanations for alerts and actions can enhance user trust and understanding.
5. **Threat Intelligence Integration:** Incorporating threat intelligence feeds and information sharing mechanisms can enrich the model's knowledge of the evolving threat landscape, ensuring it remains current and well-informed.

In conclusion, our RNN-based malware analysis model is a promising step forward in the field of cybersecurity. Its ability to detect advanced and previously unknown malware threats positions it as a valuable tool in safeguarding digital assets. Future work will focus on fine-tuning the model, expanding its application, and further enhancing its capabilities in the ever-evolving landscape of cybersecurity. This project serves as a foundation for proactive and robust malware defense.

## References

[1] “What are Recurrent Neural Networks? | IBM.”

[https://www.ibm.com/topics/recurrent-neural-networks#:~:text=A%20recurrent%20neural%20network%20\(RNN,data%20or%20time%20series%20data.](https://www.ibm.com/topics/recurrent-neural-networks#:~:text=A%20recurrent%20neural%20network%20(RNN,data%20or%20time%20series%20data.)

[2] “Introduction to Recurrent Neural Network,” *GeeksforGeeks*, May 18, 2023.

<https://www.geeksforgeeks.org/introduction-to-recurrent-neural-network/>

[3] “Malware-Detection-using-Deep-Learning,” *Github*, Nov. 11, 2019.

<https://github.com/riak16/Malware-Detection-using-Deep-Learning> (accessed Oct. 22, 2023).

[4] Vinesmsuic, “Malware Detection using DeepLearning,” *Kaggle*, Mar. 30, 2021.

<https://www.kaggle.com/code/vinesmsuic/malware-detection-using-deeplearning>

[5] “Learn Python, Data Viz, Pandas & More | Tutorials | Kaggle.”

<https://www.kaggle.com/learn>

[6] “Working with RNNs,” *TensorFlow*.

[https://www.tensorflow.org/guide/keras/working\\_with\\_rnn](https://www.tensorflow.org/guide/keras/working_with_rnn)