ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΜΕΤΑΠΤΥΧΙΑΚΟ ΔΙΠΛΩΜΑ ΕΙΔΙΚΕΥΣΗΣ (MSc)

στην ΑΝΑΠΤΥΞΗ & ΑΣΦΑΛΕΙΑ ΠΛΗΡΟΦΟΡΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

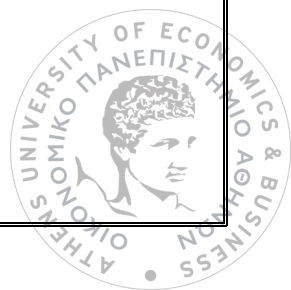
ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

“Ανάπτυξη αλγορίθμου βαθιάς μηχανικής μάθησης για αναγνώριση εντολών και πρόβλεψη συμπεριφοράς κακόβουλου λογισμικού”

Χατζόπουλος Γεώργιος

F3311908

ΑΘΗΝΑ, ΝΟΕΜΒΡΙΟΣ 2020



Περίληψη

Η ραγδαία αύξηση των τεχνολογικά προηγμένων κυβερνοεπιθέσεων καθώς και ο αυξανόμενος αριθμός των νέων κακόβουλων λογισμικών, αποτελούν πλέον ένα μείζον θέμα που έχει να αντιμετωπίσει ο τομέας της ασφάλειας. Άγνωστα κακόβουλα λογισμικά, τα οποία δεν έχουν ταυτοποιηθεί από συστήματα ασφάλειας και χρησιμοποιούνται σε επιθέσεις καθιστά δύσκολο τον εντοπισμό τους αλλά και την λήψη κατάλληλων μέτρων με σκοπό τον αποκλεισμό τους από το εκάστοτε υπολογιστικό σύστημα. Η εκ των προτέρων γνώση της συμπεριφοράς κάποιου κακόβουλου λογισμικού καθιστά τον εντοπισμό αλλά και την πρόβλεψη της μελλοντικής συμπεριφοράς του, ένα πολύ χρήσιμο εργαλείο προκειμένου να ληφθούν αντίμετρα για τον περιορισμό της βλάβης σε κάποιον υπολογιστικό πόρο. Στη παρούσα διπλωματική δίδεται ιδιαίτερη έμφαση στον τρόπο με τον οποίο μπορούμε να προβλέψουμε με τη βοήθεια της μηχανικής μάθησης, την μελλοντική συμπεριφορά κάποιου κακόβουλου λογισμικού βασισμένοι σε μία σειρά εντολών που ήδη έχει εκτελέσει στο υπολογιστικό σύστημα που έχει μολύνει. Σε πρώτη φάση εκπαιδεύουμε το νευρωνικό δίκτυο με μία σειρά εντολών που έχουν χρησιμοποιηθεί από το malware, προκειμένου να μπορέσει να αναγνωρίσει σχέσεις μεταξύ αριθμών καθώς και την σειρά εκτέλεσης τους. Σε επόμενο βήμα, λαμβάνοντας υπόψιν τις ήδη υπάρχουσες εντολές προσπαθεί να προβλέψει ποια εντολή μπορεί να επακολουθήσει, κατασκευάζοντας κάθε φορά μία ν-άδα αριθμών ως μία πιθανή εντολή εκτέλεσης. Τέλος κάνουμε εκτίμηση των αποτελεσμάτων που λαμβάνουμε από το νευρωνικό σε σχέση με την αναμενόμενη εντολή.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Μηχανική μάθηση/ Κυβερνοασφάλεια

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: νευρωνικά δίκτυα, βαθιά μάθηση, πρόβλεψη, κακόβουλο λογισμικό

Abstract

Rapid development of technologically advanced cyber-attacks as well as the increasing number of new malwares, comprise a challenging issue cybersecurity has to face. Unknown malware which has not been identified yet from security systems are vastly used to the new era cyber-attacks making their tracing and containment a difficult situation to handle from every information system they infect. A priori knowledge of how a malware behaves into a system gives us a great advantage in order to trace it and to predict its future steps. Thus, we can more easily decide which counter measures should be taken to prevent any future damage to the system. The objective of this diploma thesis is to accent the benefits of machine learning, and the way it can help us identify and predict future steps of a malware given the already executed commands. First, we train the Recurrent Neural Network to identify and learn relations between the given malware commands. Next step is to predict the next incoming command that malware is going to produce. Prediction part involves the construction of an array which represents the next possible command. The evaluation result is produced by command hit ratio; specifically, how many predicted commands are as same as the expected ones.

SUBJECT AREA: Machine Learning/Cybersecurity

KEYWORDS: neural networks, deep learning, prediction, malware

ΕΥΧΑΡΙΣΤΙΕΣ

Θα ήθελα καταρχήν να ευχαριστήσω το διευθυντή του μεταπτυχιακού προγράμματος σπουδών κύριο Δημήτριο Γκρίτζαλη για την οργάνωση του προγράμματος και τη καθοδήγηση σε όλο το διάστημα φοίτησης. Σημαντικό ρόλο στην εκπόνηση της παρούσας διπλωματικής εργασίας διαδραμάτισε ο επιβλέποντας της, κύριος Στεργιόπουλος Γιώργος που μου έδωσε την ευκαιρία να ασχοληθώ με αυτό το θέμα. Τον ευχαριστώ θερμά για την άψογη καθοδήγηση, συνέπεια και εξαιρετική συνεργασία που είχαμε καθ' όλη τη διάρκεια ενασχόλησης μας με την εργασία.

Θα ήθελα να ευχαριστήσω τους γονείς μου για την υπομονή τους, την ηθική συμπαράστασή τους που μου προσέφεραν απλόχερα από την πρώτη μέχρι και την τελευταία μέρα ως μαθητής. Τέλος, θα ήθελα να ευχαριστήσω τον αδερφό μου Νικόλα για τον οποίον αποτελώ πηγή έμπνευσης και απ' όπου αντλώ δύναμη να συνεχίζω τη προσπάθειά μου.

Contents

Chapter 1

1 Introduction

Nowadays cyber-attacks based on malware are increasing vastly every year, hitting more and more businesses every day. The way malware is constructed is more advanced and sophisticated, which makes it even more difficult to understand how it works and what potential damage could cause to an infected system. There are a lot types of malware including the polymorphic ones. Polymorphic malware is a type of malware that constantly changes its identifiable features to evade detection. Thus, it is harder not only to identify but to trace and detect patterns of such malware in a system. The most common types of malware, viruses, and worms are known for the way they penetrate a system, rather than any specific type of behavior. A computer virus is a piece of software that embeds itself in executable software and when it is triggered the virus starts to replicate itself infecting system's programs. Inside the infected system the whole replication and infection process is undergoing without user's knowledge and consent. Another major category is worms. Worm is a malware software which uses an initial computer to spread and afterwards infect other computer systems inside a network. Its main function is to recursively replicate itself in order to exponentially grow and finally spread and infect more and more computer systems. As we can assume from the previous definitions human factor is a major piece on cyber-attack puzzle because malware and viruses demand the user privileges to be executed from an infected software. In many advanced attacks the unknown virus cannot be identified from the security vendor and it becomes difficult to perfectly defend terminal from attacks and demands for after infection countermeasure are increasing.

1.1 Machine Learning with Malware?

Over the past decades *machine learning* has been used in a wide range of applications from virtual personal assistants ready to answer any question you may have, to online fraud detection protecting even against money laundering. Naturally, cybersecurity is not out of the application areas of machine learning. There is a general belief among cybersecurity experts that antimalware tools and systems powered by AI and ML will be the solution to modern malware attacks. According to Google Scholar, the number of research papers published in 2018 is 7720, a 95% increase with respect to 2015 and a 476% increase with respect to 2010. (Daniel Gibert, 2020). This can be explained since many ML algorithms can give great results for malware-pattern detection and for malware classification determining whether a given a set of commands corresponds to a malware or not.

1.2 Scope of this thesis

Motivated by the unlimited capabilities of Machine Learning including Neural Networks, the present thesis focuses on analyzing API calls from a malware, learn the structure of commands that malware uses and try to predict the next incoming commands. The research focuses on RNNs and their effectiveness while learning long-term dependencies between malware's different commands, which will have a catalyst impact on the prediction stage. In Machine Learning language RNN stands for Recurrent Neural Network and it is a class of artificial neural networks where connections between nodes form a directed graph along a temporal sequence. This allows RNN to have temporary dynamic behavior. RNNs due to the fact that they are recurrent, they have inherited from feedforward neural networks the ability to keep an internal state that works like a memory to process variable length sequences of inputs. Thus, RNNs can be applicable to many tasks such as speech recognition or handwriting recognition and sequence prediction models. One of the main targets, was to examine how well the model of the RNN would behave on pseudo-random data which mimic a potential malware.

1.3 Thesis Structure and planning

The chapters are classified as follows:

- Chapter 1: Introduction and definition of Thesis scope and final goals
- Chapter 2: State of the art, including most recent researches upon malware analysis with machine learning
- Chapter 3: Analyzing techniques, general idea of the RNN we built and overview of the model we developed.
- Chapter 4: Presentation of our experiment, total evaluation, the results, and future possibilities of the project

Chapter 2

2 State of the art

Many studies have shown the effectiveness of machine learning in the field of malware detection. In this section we are going to present previous researches of malware detection and classification techniques and advanced methods based on Neural Networks.

A. Malware detection based on Windows Application Programming Interface

Detection of malware inside an infected system can be done with two methods static and dynamic. Static malware analysis does not require execution of the malware in order to be analyzed, but the core idea proposes method such as string search through the program to understand its functionality or even inspecting file headers. On the contrary, dynamic analysis includes inspection while program is running. Static analysis has a certain drawback which is the lack of effectiveness against more sophisticated malware such as polymorphic malware. Furthermore, obfuscation techniques can be used to bypass static analysis, a factor that makes programming code hard for the static analysis detector to understand whether the code is malicious. There are many obfuscation techniques which include encrypting parts of the initial code, hiding useful metadata and also name alternation for both classes and variables.

On the other hand, dynamic analysis monitors malware's code behavior in real-time. Dynamic analysis contains features extraction from network behavior, registry changes, system calls and memory usage. Through API calls we can create a chain of sequences gathering calls from executing programs to define whether it's behavior can be malicious or not. The sequence of API calls can provide meaningful expressions that support and assist in better understanding of a malware. There are several ways to translate each API call. In many researches they can be transformed into regular expressions defining certain rules of determining if a call is malware or not. In NLP, a standard way to define sequences is to interpret them to numerical input where machine learning algorithms can work with. (Serif Bahtiyar, 2019)

The representation of each sequence can be done in many ways. A proposed one is through Markov chains. In this way, we can use Markov chains to produce behavioral models for malware and normal processes that are running currently into a system (goodware). The resulted models can be considered as our representative features, which describe malware and goodware. (Eslam Amer, 2020)

In this thesis we will focus on dynamic API calls and their analysis in order not only to define if a certain command is coming from a malware but also try to predict the

next incoming commands from a malware. The API calls can be gathered into a large dataset where they can be learned more easily from a Neural Network. The RNN will try to understand the correlation between commands combining pattern recognition techniques through layers of MLP and LSTM.

B. Malware Detection Method with Deep Neural Networks Classification

In most papers, Deep Neural Networks have been used to detect and define whether a sequence of commands is a malware or not. In machine learning language this process is called *classification*. It is considered as a task of machine learning algorithms that learn how to assign a class label to examples from the domain problem. An example is classifying e-mails as 'spam' or 'ham' (not spam). Machine learning researchers have proposed several models of classifiers for malware classification such us, decision trees, logistic regression, neural networks and currently RNNs.

Another different approach is to use CNNs. CNN stands for Convolution Neural Network and is the current state-of-art neural network for image classification problem. This technique spectates malware classification from a different point of view as it suggests visualizing malware as image. Given a malware binary file each of its components it is converted to an equivalent decimal value (e.g. For the binary vector [1 1 1 1 1 1 1 1] we have the corresponding decimal value of [255]) which later be used to compose a grayscale image of the malware. To be more precise each one of these decimal values will represent a single pixel on the greyscale image. To train this model we have predefined categories(families) of different malware names e.g. trojan, virus etc. and to optimize classification loss, cross-entropy loss is chosen for training. This sophisticated architecture takes advantage of the effectiveness of CNNs on image classification and thus malware detection can be represented as such. (Mahmoud Kalash, 2018)

Another model used for malware process detection and classification is the one that combines two types of Deep Neural Networks (DNN). This architecture contains a hybrid model of a RNN and a CNN. Firstly, the RNN will get as input recorded API calls and try to extract the feature of process behavior. This functionality is the same with NLP models, where sentences consists of words, correspondingly API calls contain features of malware behavior. Thus, we expect from the RNN to learn correlation between API calls. The feature vectors extracted from the RNN will be transformed into grayscale image. This image potentially contains various local features which represent process activities. Next step is to apply CNN to classify

these images by training these local features. Finally, The CNN trains imaged features of malware and benign processes to create a classifier. (Shun Tobiyama, 2016)

In our thesis, we present a hybrid model which contains a combination of three Deep Neural Networks. The first slice of the model is a MLP (Multi Layered Perceptron) which tries to learn the correlation between the integers we passed as dataset. The outcome of the MLP will be used as input for the RNN. This RNN contains an LSTM layer capable of holding long term dependencies between the appearance of each row and thus its versatility can be vital to predict next incoming sequences. Finally, the last piece of this architecture is the output MLP layer which is used to learn the dependencies between the numbers of the output of the LSTM layer and also update the weights of the existing model for better accuracy on prediction stage.

Chapter 3

3.1 Dataset selection

To run our experiments properly, we need a dataset that has already transformed an API sequence to corresponding integers as it is time savior and more efficient for the algorithm to learn at training phase. For instance, we can index the function call *deviceiocontrol* with the number 40. Thus, a row from our dataset consists of a sequence of integers which corresponds to a certain function malware call. Each row consists of a sequence of 20 numbers which will

55	17	29	9	44	76	51	55	7	74	25	28	67	70	27	37	78	30	85	75
15	8	77	2	7	5	34	72	1	17	33	29	57	72	42	73	75	29	15	43
5	15	66	7	77	62	74	41	9	22	75	68	43	21	20	3	42	1	45	25
57	55	78	6	11	18	64	58	2	71	24	74	17	20	11	80	47	24	72	40

Figure 1: Four first rows from our dataset

be extremely useful for training stage, defining layers of a RNN model.

The following encoding has a certain purpose, as machine learning algorithms seem to perform better with integers (applicable with float numbers as well). For our experiment conduction we have chosen to use 2 different kinds of datasets. The first one, consists of pseudorandom integers we have produced in the lab to test how our model will behave on totally random data. The second one, consists of sequences of gathered Windows API calls that have been executed by a malware inside an infected system.

In the first case, generated numbers are between [0,80]. Each row consists of 20 unique integers, with no duplicate integers in a row, because there is no occurrence of the same command inside a sequence. In the second case, integers range from 1 to 112. Each row of the

second dataset consists of 100 integers where we can have duplicate values. In this case, we consider that we may have a repetition of the same command.

Encoding selection for the dataset was not randomly chosen. Integer encoding is vastly used on NLP (Natural Language Processing) in order to match words, letters to a single index. The results of trained neural networks with integer encoding are quite remarkable, including music generation, language to language translation and even phrase generation. Correspondingly, in our project we can mimic API call generation to music or phrase generation.

As next step, to achieve even more better results we will scale the dataset and transform it with multi-hot encoding. This will be the final shape of the data before they enter training phase.

3.2 Model Description

Recurrent Neural Networks have been applied to a lot and different problems including our case of sequence prediction. A categorization of RNNs:

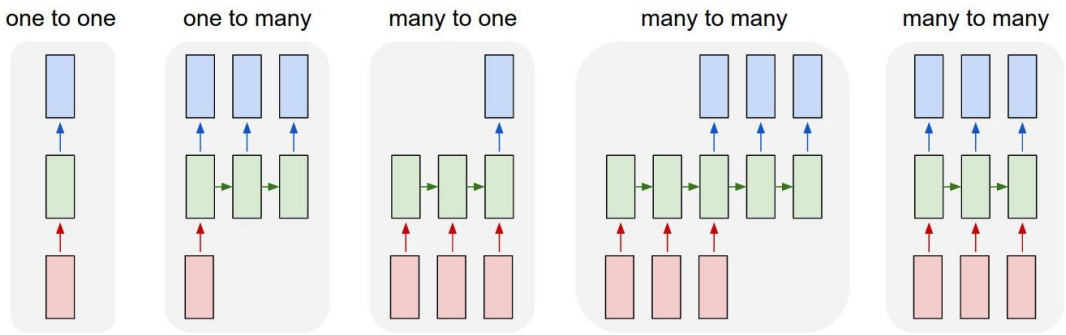


Figure 2 RNN models (https://towardsdatascience.com/an-introduction-to-recurrent-neural-networks-for-beginners-664d717adbd)

- *One-to-one model*: This type of model is the vanilla mode of processing without RNN, from fixed-sized input to fixed-sized output. RNNs initial state gets one time step each time and produces one output for each input. (Brownlee, Machine Leaning Mastery, 2019)
- *One-to-many model*: This type of model produces multiple outputs for one input. It can be applied to produce description of an image, where image is the input, and the caption would be the output describing the picture. (Brownlee, Machine Leaning Mastery, 2019)
- *Many-to-one model*: This model produces one output value after receiving multiple time-steps as input. It can be used for classification purposes e.g. to examine whether

an email is spam or ham based on the words in the sentence. (Brownlee, Machine Leaning Mastery, 2019)

- *Many-to-many model*: A many-to-many model produces multiple outputs correspondingly to the number of inputs. In predicting stage, we give multiple time steps and we expect the output to be at least bigger than one sequence. The size of the output is not fixed since it can be less than the input sequence e.g. on translation models a sentence in English could have more words than the corresponding sentence in French. (Brownlee, Machine Leaning Mastery, 2019) . The model we developed belongs in this category of RNNs. In particular, our model gets as an input a single row or multiple rows of the dataset each one consisting of 20 integers and then we want the output of the algorithm to be a 20-integer array. So, in this scenario we have a Many-to-many model where the input and the output length are the same number 20.

3.2.1 Long Short-Term Memory (LSTM)

One special category of Recurrent Neural Networks are LSTM networks. LSTMs can be applied in prediction models where the sequence of data is important to predict next data and it performs much better than a vanilla version of a standard RNN model. It stands for Long Short-Term Memory and it is vastly used for its capabilities of learning long-term dependencies. (Ralf C. Staudemeyer, 2019) A main problem that a standard RNN has to face is the problem of Long-Term dependencies. In many cases, where we need more context especially in the field of NLP, is necessary that our RNN holds recent and past information in order to construct a new one. Consider the following sentence 'I am from Greece…my mother tongue is *Greek'*. In this case what we want to achieve is to train the RNN model to predict the word *Greek* of the sentence. As human beings and based on the information of the sentence we can easily infer that the last word is Greek. In machine learning world the gap between holding information and try to predict new one based on the previous, is filled with LSTMs which seems to be the missing puzzle piece, helping RNNs to achieve such remarkable results. (Olah, 2015)

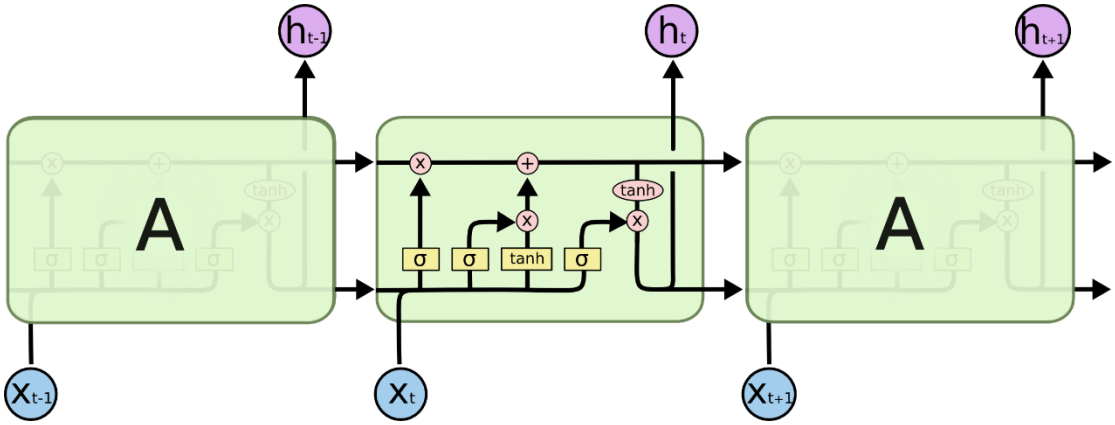


Figure 3 LSTM anatomy

LSTMs structure is designed to solve the long-term dependency problem and the secret lies at their sophisticated architecture. Contrary to standard RNNs where the repeating module contains a single tanh layer, LSTMs contains four neural network layers working alongside.

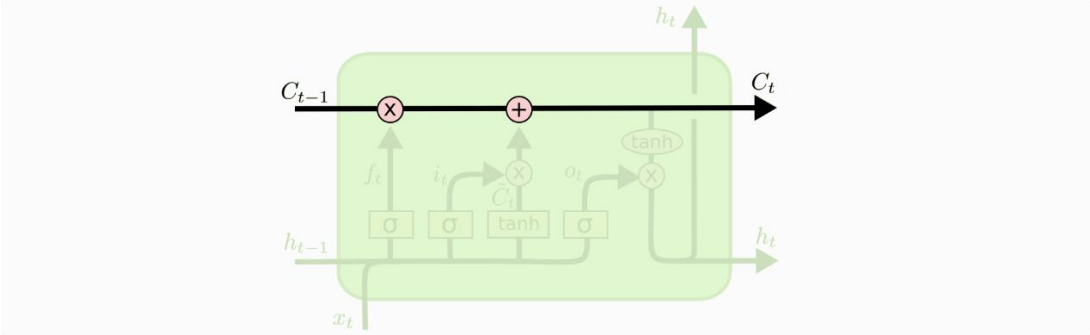


Figure 4 Cell state

The cell state of LSTM is the key to its success as it runs the cell and makes it for the information to flow along deciding what information to pass through, to alter or to throw away from the cell state. Responsible for such decisions is the first sigmoid layer called the *forget gate layer*. The forget gate outputs two possible each one with a certain meaning. If the output is 0 it signals cell state to discard the incoming value, and in the other case when output is 1 it signals cell to preserve the value. (Olah, 2015)

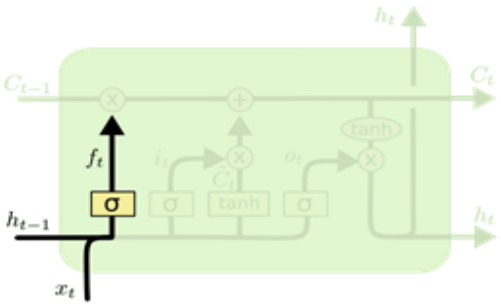


Figure 5 Forget Gate

Next step LSTM decides what information will be stored in the cell. Sigmoid and tanh layer play an important role for updating and adding new values in the cell. The first one will choose the update values and the last one will create a vector with values that will be added into the cell. Finally, combining the output of the two a cell update is created. In order to update the old cell LSTM multiplies the old state with the output of the forget gate and then it adds the outcome of input sigmoid layer and tanh layer. (Olah, 2015)

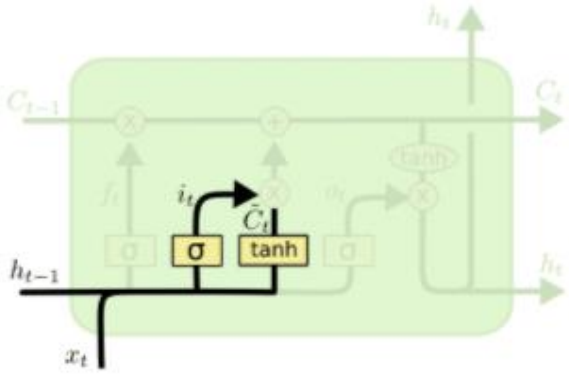


Figure 6 Input sigmoid σ layer and tanh layer

Proceeding to the final step, LSTM needs to decide what exactly its output is going to be. The outcome will be decided from the final sigmoid layer which is responsible for deciding which parts of the cell will go through to the final output. After applying that layer cell state passes through tanh layer and it is multiplied with the output of the sigmoid gate. (Olah, 2015)

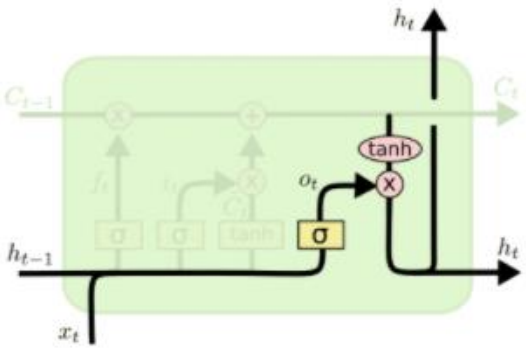


Figure 7 LSTM output

For programming purposes Keras library in Python offers the implementation of LSTM layers which we used to build our algorithm.

3.2.2 Multi-hot encoding vs One-Hot Encoding

One-Hot encoding

In many machine learning algorithms, we come across with the term ‘one-hot encoding’. One hot encoding is a process by which categorical variables are converted into a form that could be provided to machine learning algorithms to do a better job in prediction (Vasudev, 2017) . As a matter of fact, encoding is proven to one good way to have better results in both training and prediction phases. One hot encoding depicts data as binary vectors and more precisely it maps every variable with a 0 except the index of the variable, we are interested in which is marked with 1. The reason encoding is vastly used is because it allows data to be less complex and makes things easier for machine learning algorithms because cannot work with categorical data straightforwardly. For instance, we have a sequence of labels ‘TV’, ‘Computer’, ‘TV’. We can ‘TV’ an integer value 0 and ‘Computer’ the integer value 1. This technique is called binary encoding. Later on, after predicting stage we will reverse the matching and binary will be represented as labels again. Thus, in both the training and prediction phase the algorithm will use encoded binary data. As next step, we make a binary vector to represent each integer value we gave to our data. The vector for each value will be a length of 2 since we only have 2 possible values. As a result, the ‘TV’ label will be represented as a binary vector [1, 0] where the zeroth index is marked with a value of 1. Correspondingly, ‘Computer’ label encoded as a 1 will be depicted as [0, 1]. (Brownlee, Machine Learning Mastery, 2020)

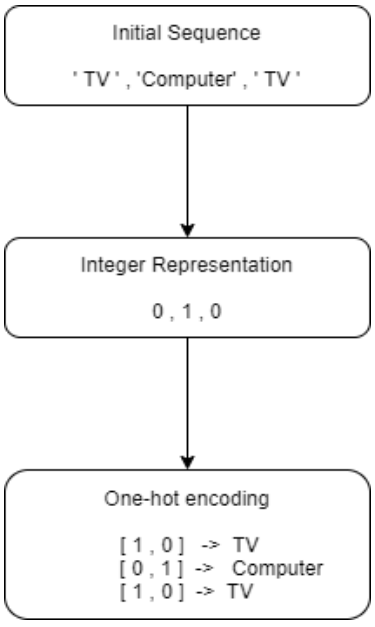


Figure 8 One Hot encoding procedure

Multi-Hot encoder

On the contrary, multi hot encoder is based on a different approach of labels and data than one hot encoding. In particular, is an artificial vector created on machine learning processes in order to represent categorical variables in a multi-dimensional space by encoding them into numerical values. In this way, when we try to encode a sequence of labels, we do not have to encode each label individually but instead we combine all of them in one output. For instance, if we have the sequence of numbers 3,5,10 we will be creating a binary vector to express those 3 numbers in one sequence. Encoder's output for the above input would be 00010100001. Number 3 corresponds to the 1 on the 3rd index, number 5 to the 1 on the 5th index and finally 10 to 1 in 10th index. If we have chosen to do it with one hot encoding the outcome would be [0,0,0,1,0,0,0,0,0,0,0] for number 3, [0,0,0,0,0,1,0,0,0,0,0] for 5 and [0,0,0,0,0,0,0,0,0,0,1] for number 10. The difference between the two encoding methods is clear, particularly if we consider the fact that for an integer array of 20 numbers with multi hot we only have 1 binary vector while with one hot we have 20 different binary vectors.

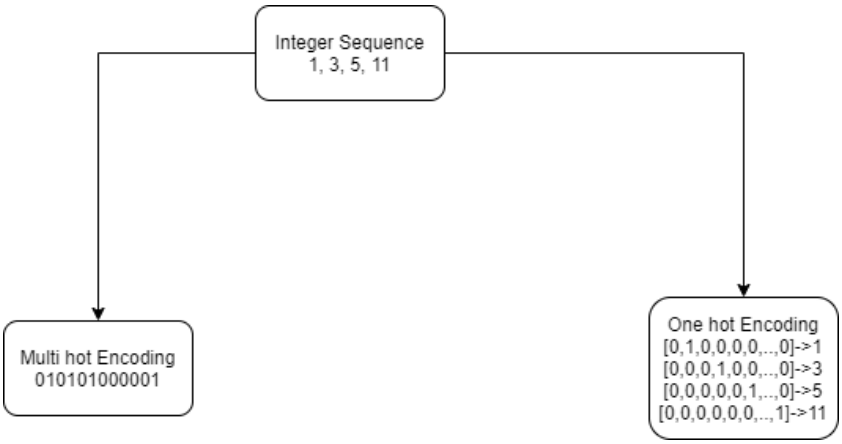


Figure 9 Multi hot vs One hot

3.2.3 Keras library

The most useful tool we had in our hands while developing the algorithm was Keras library. Keras is an open source library for Python specifically developed to help engineers, students and researchers build Artificial Intelligence algorithms, explore machine learning capabilities in a fast way. For our purposes we used Keras preprocessing capabilities which contained the already developed layers of LSTM, MLP. (Keras, 2020)

3.3 Proposed method

In this chapter, we will analyze our proposal for malware detection and malware sequence generation. The proposed method applies 3 Neural Networks MLP, LSTM, MLP in 3 stages. The first stage uses an MLP layer to try and learn relations between the numbers in the input

sequence. In the second stage we use LSTM, to get the output of the MLP and try to combine already gained knowledge on each number and update the weights in the algorithm. Finally, the last MLP will transform the LSTM output to the predicted sequence.

3.3.1 Model Overview

The overview of our model can be found below in Figure 10 '*Big Image of our model'*. Model's architecture is composed of various functions as Process Dataset, and these functions consist of multiple operations.

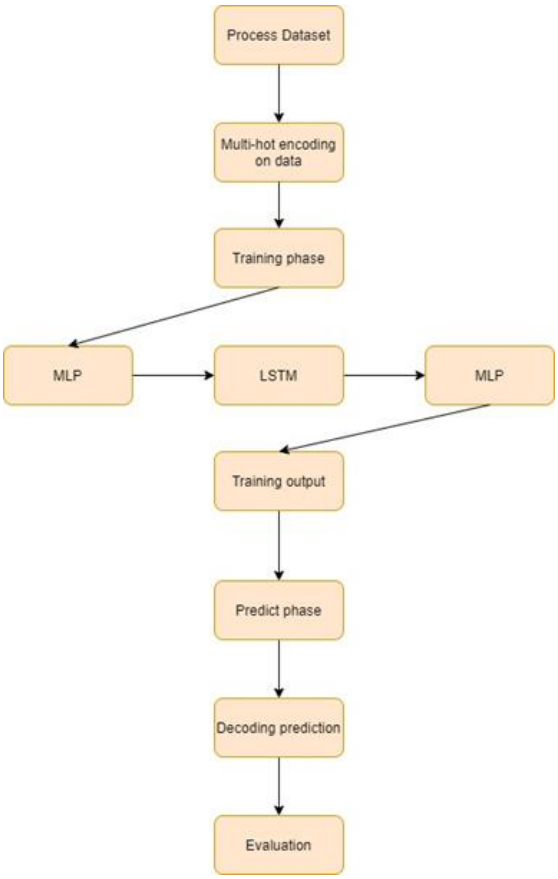


Figure 10 The big image of the model

When we start with *data processing*, we generally try to load our already integer encoded datasets from .csv format in data frames which speed up the data manipulation stage. For that reason, python offers a very fast and powerful library called *Pandas.* It offers functions that import .csv files to data frame format directly. Pandas Data Frames is an excel-like data structure with labeled axes (rows and columns). Height and width of the data frame will be defined from the rows and columns of the original dataset on the .csv file. In column field since we have a fixed number of sequences length, we are sure that the number will be 20.

Next step is to pass data through *multi-hot encoding*. As we explained in section 3.2.2, we emphasized the fact that binary vectors can offer better results as far as training and prediction

evaluation is concerned. The created data frame will be a representation of each row in binary form.

Before we proceed in the *training phase*, we need to define all the essential parameters for the model to train properly. Firstly, we have to split our dataset into training and testing data. The split ratio is defined as 80% for training and the rest 20% for testing. The created data frame will be divided into two smaller data frames, one that will contain the 80% of the original dataset and the second will contain the rest 20%. In this way, at the end of training there is still data which algorithm has not seen through the training process. In machine learning algorithms, for a model to be trained we need to have the input data and the target data. Originally, target data contains features or labels and expresses what the model is trying to learn based on original data. Supervised machine learning algorithms need to know in advance what outcome we expect otherwise the training process would not have any sense. Furthermore, if target data are not clear to the algorithm, then the outcome would not be the expected one and thus it will not be capable to learn and later to predict. In our case, since we need to predict sequences of numbers and not a single number each time we have to set as target the next rows of the input dataset. This means that both input data called data_X and target data called data_Y will contain the same number sequences with a slight difference. Data_X will contain exactly the same sequences as training dataset. However, in data_Y the first row will be data_X second row, in second row will be data_X third row etc. and this happens because we want each time to predict the next row. Thus, training algorithm will try to match the first row of data_X with first row of data_Y.

56	29	41	80	13	59	61	33	27	64
20	76	47	31	1	25	8	6	40	51
62	56	61	5	11	27	28	4	53	16
60	63	5	25	48	11	52	51	61	70
74	5	73	56	24	23	77	55	9	7
46	23	45	14	54	2	62	19	52	73

Figure 11 Six first rows of data_X

20	76	47	31	1	25	8	6	40	51
62	56	61	5	11	27	28	4	53	16
60	63	5	25	48	11	52	51	61	70
74	5	73	56	24	23	77	55	9	7
46	23	45	14	54	2	62	19	52	73
11	14	69	7	60	70	74	50	13	80

Figure 12 Six first rows of data_Y

As we can see in the figures 11,12 algorithm will try to predict the first row of data_Y given the first row of data_X and so on. In this way, we model the many to many neural networks we mentioned at section 3.2. At this point we can enhance training phase by making the model more dynamic. As we mentioned earlier, training works by learning data_X and trying to predict data_Y one by one. A more dynamic approach would be to predict data_Y by learning more than one data_X rows per training batch. This can be simply done by adding a look back parameter to be set in a number defined before training stage. If lookback is 6, it means that in order to predict one row of data_Y should firstly learn 6 rows from data_X. This method looks like the sliding window, with window size = 6, where data_X and data_Y are sliding with 1 row at each iteration.

7	56	29	41	80	13	59	61	33
64	20	76	47	31	1	25	8	6
42	62	56	61	5	11	27	28	4
53	60	63	5	25	48	11	52	51
60	74	5	73	56	24	23	77	55
71	46	23	45	14	54	2	62	19
56	11	14	69	7	60	70	74	50
71	17	55	41	40	23	21	12	25
24	10	62	21	14	63	53	48	79
9	38	25	33	72	53	49	70	47
57	63	7	77	1	32	39	5	72
20	34	18	66	6	30	38	8	61

Figure 13 In green boxes rows of dataX and in red the expected outcome dataY. Initial state configuration with lookback = 6

7	56	29	41	80	13	59	61	33
64	20	76	47	31	1	25	8	6
42	62	56	61	5	11	27	28	4
53	60	63	5	25	48	11	52	51
60	74	5	73	56	24	23	77	55
71	46	23	45	14	54	2	62	19
56	11	14	69	7	60	70	74	50
71	17	55	41	40	23	21	12	25
24	10	62	21	14	63	53	48	79
9	38	25	33	72	53	49	70	47
57	63	7	77	1	32	39	5	72
20	34	18	66	6	30	38	8	61

Figure 14 Data_X and Data_Y slide by 1 row

7	56	29	41	80	13	59	61	33
64	20	76	47	31	1	25	8	6
42	62	56	61	5	11	27	28	4
53	60	63	5	25	48	11	52	51
60	74	5	73	56	24	23	77	55
71	46	23	45	14	54	2	62	19
56	11	14	69	7	60	70	74	50
71	17	55	41	40	23	21	12	25
24	10	62	21	14	63	53	48	79
9	38	25	33	72	53	49	70	47
57	63	7	77	1	32	39	5	72
20	34	18	66	6	30	38	8	61

Figure 15 Data_X and Data_Y slide by 1 row

At this point, after dataset processing, we are entering the model's *training phase*. A major factor for a model's training is right configuration and coordination of Neural Networks. The model consists of 2 MLP and one LSTM network. At this point we need to focus on the proper data reshaping to fit in each layer. MLP gets a 2D input shape and LSTM needs a 3D input. Thus, we must reshape both data_X and data_Y from 2D to 3D. Even though, MLP gets a 2D input we can choose to pass two dimensions from a 3D input. In MLP the input shape consists of the batch size, and the number of features our input dataset has. Batch size in machine learning is defined as the number of samples(rows) that will be propagated through the network. The next layer is the LSTM. In LSTMs we have a 3D input shape where the first dimension corresponds to number of samples, second dimension to time steps and third dimension represents number of features. With the term time step, we mean the number of steps that the neural network needs to look back to previous sequences in order to predict the next one. Finally, the output of LSTM will be passed into the final MLP layer. This last step will produce the predicted sequence.

Afterwards we proceed to model's *prediction phase*. To generate new predicted sequences, we are going to use the test dataset we introduced at the begging of the overview. For prediction to work properly we have again two possible ways. First one, works like a one-to-one model. We feed predict function with one sequence at a time and we expect one sequence as output. The second way is to give the trained neural network an input of many sequences e.g. 6 rows of the dataset and expect as output one sequence. It works like a many-to-one model with the advantage of getting more information as input.

Final step is model *evaluation*. At this point, we compare the output of the prediction phase and the expected results. Each result corresponds to a hit ratio between each prediction and expected sequence. If ratio is above or even to 50% it means that the output has correctly predicted the half number of the expected sequence and we consider that to be a success hit.

Chapter 4

4.1 Results of the experiment

In the last segment of our thesis we will represent an example of the model we used to get a better understanding and focus more on its details.

4.1.1 Training model

```
def model(dataX, dataY, t_X, t_Y):
    """

    :type dataX: Includes the input data
    :type dataY: Includes target values

    """
    model = Sequential()
    # First model layer with MLP
    model.add(Dense(512, input_shape=(20, 80), activation='relu'))
    # feeds output to LSTM
    # LSTM input values (rows,columns,features)
    model.add(LSTM(1024, return_sequences=True))
    # Final layer with specific output/ trying to predict the next incoming sequence
    model.add(Dense(80, activation='sigmoid'))

    # model.build(data.shape)
    model.compile(loss='binary_crossentropy', optimizer=keras.optimizers.Adam(lr=0.0001))
    model.summary()

    model.fit(dataX, dataY, batch_size=80, epochs=1000)
```

1 Training Stage: Model train function

To start training process, we need to define a function equipped with the proper layers to begin data processing. The model we chose to go with is Sequential model. From Keras documentation (KerasDevTeam, n.d.) sequential model is used when layers are stacked one after the other, and also when the developed model has one input and one output tensor. In our case, we have one MLP tensor as input and another one as output.

As next step we add to the model its first layer, an MLP layer. In Keras MLP is depicted with Dense function. Then, we configure the layer parameters to work properly. The number of neurons we have is 512, the input shape is [20,80], and the activation is ReLU. The number of neurons was chosen after running the experiment many times with different numbers each time and the conclusion was that 512 has given the best results. Input shape takes 2 parameters. The first one is for the number of features each sequence of the input has and the second one is the number of timesteps to lookback to predict the next value. In our case the number of features per row on the dataset is 80 and also the lookback is set to 20. Finally, activation is configured by selecting a function to give us the output of a node. ReLU

(Rectified Linear Unit) is a linear function and ranges from 0 to infinity which means negative values are rounded to zero.

Next layer is the LSTM. The main job is to get the output of the previous MLP layer and update the weights between the nodes. Here the units are set to 1024, and the return_sequences parameters is set to True. When this last parameter is set to True, it indicates that the output of the LSTM layer is going to be a sequence.

The last layer is an MLP layer which takes the output from the LSTM and transforms the final sequence to be an 80-length binary vector correspondingly to activation function we used. In contrast with first MLP, the last one has a sigmoid activation function. The selection of this specific function has a certain purpose to transform the output to a binary vector. Sigmoid turns the output of LSTM to 0 or 1.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 20, 512)           41472
_________________________________________________________________
lstm (LSTM)                  (None, 20, 1024)          6295552
_________________________________________________________________
dense_1 (Dense)              (None, 20, 80)            82000
=================================================================
Total params: 6,419,024
Trainable params: 6,419,024
Non-trainable params: 0
```

2 Model Summary

After training stage, the output will be a sequence of 80 binary numbers each time a batch is processed. The final step to run the model is to define batch size and number of epochs. Batch size is the number of samples from the input, that algorithm would process at each iteration e.g. Consider a model with batch size = 80, epochs = 640 and input data frame = 8000. Firstly, we divide the data frame with the batch size to check in how many steps our data frame will be processed. In one epoch the algorithm will feed each block of 80 sequences to the training model until we reach 8000. So, at the first epoch the algorithm will iterate the dataset 100 times taking each time 80 rows until it reaches the end of the data frame. Second epoch will repeat the same process as first and so on until algorithm reaches epoch 640. At

each epoch we have the loss function which gives an image of how good our algorithm performs.

```
163/163 [==============================] - 6s 38ms/step - loss: 0.0128
Epoch 635/640
163/163 [==============================] - 6s 39ms/step - loss: 0.0126
Epoch 636/640
163/163 [==============================] - 6s 39ms/step - loss: 0.0124
Epoch 637/640
163/163 [==============================] - 6s 38ms/step - loss: 0.0122
Epoch 638/640
163/163 [==============================] - 6s 38ms/step - loss: 0.0120
Epoch 639/640
163/163 [==============================] - 6s 39ms/step - loss: 0.0118
Epoch 640/640
163/163 [==============================] - 6s 39ms/step - loss: 0.0116
```

3 Loss at last steps of the algorithm

As we can see our model has achieved a small loss which indicates that it is learning properly the correlation between train and target data. The loss function we have chosen is binary cross entropy since we have configured our algorithm to work with binary vectors. Finally, the last parameter is the optimizer. In our case, Adam optimizer is used to update weights on the training data. For that purpose, it uses learning rate which is configured to 0.0001 to handle data overfitting which remain unchanged during the training phase.

4.1.2 Prediction stage

After training we start prediction stage. First step is to take test data and pass them trough predict function. The outcome will be a binary vector which is going to be decoded to integer values. At this point we invert the encoded values a process we did at the start of the experiment with multi hot encoding. Each binary vector is going to be matched at its original integer value to construct the number sequence.

```
[[1 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
  1 0 0 1 1 0 1 0 1 1 0 1 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 1 0 1 0
  1 1 0 0 0 1 0 0]]
[[ 1  9 12 17 20 34 37 40 41 43 45 46 48 57 61 69 71 73 74 78]]
```

Figure 16 Multi hot decoding

In figure 16, we introduce a snapshot of how multi hot decoding works after prediction stage. Every index in binary vector which position is marked with 1, it corresponds to an integer number for the output sequence.

Afterwards, we check each sequence prediction from the algorithm and compare it with the expected one giving us a hit ratio. Subsequently we can measure how well our predictions have gone. We consider a hit when at least half of the integers (≥50%) on predicted sequence are as the same as integers on expected sequence. If we feed prediction function with already seen data, the algorithm performs exceptionally good which indicates how well the training phase went. In fact, as we can see from the figure below hit ratio is 100%.

```
>Predicted [ 3 15 17 18 21 23 24 25 26 27 28 29 41 43 51 53 59 75 79 80]
>Expected= [ 3 15 17 18 21 23 24 25 26 27 28 29 41 43 51 53 59 75 79 80]
1.0
>Predicted [ 3  8 10 11 16 21 31 38 39 43 44 50 53 57 61 63 69 70 76 77]
>Expected= [ 3  8 10 11 16 21 31 38 39 43 44 50 53 57 61 63 69 70 76 77]
1.0
>Predicted [ 1  3  6 13 19 20 21 32 35 41 42 44 46 47 51 54 57 67 75 80]
>Expected= [ 1  3  6 13 19 20 21 32 35 41 42 44 46 47 51 54 57 67 75 80]
1.0
```

Figure 17 Predictions with already seen data

With completely new data, the algorithm slightly reaches the limit of the hit ratio, with an approximate ratio of 0.50. In the figure below we can see one of the 600 predictions made from the algorithm which has a ratio of 0.47. It can give us a clear image of what malware's behavior will be.

```
>Predicted [10 14 17 22 23 24 26 45 46 52 56 58 60 61 65 67 69 79]
>Expected= [ 1  6 10 12 17 22 23 33 45 47 51 53 56 58 62 63 67 69 72 80]
0.47368421052631576
```

Figure 18 Predictions with random data

4.2 Future work

Malware's power is increasing due to fact that more and more sophisticated methods have appeared over the years. A major factor that turns malware into a bog threat for any computer system is the lack of quick identification and prediction of future behavior inside a system. With our proposed way, we tried to show a different approach of how Neural Networks can solve the riddle of prediction next steps of a malware. As machine learning world keeps growing new tools more powerful and more accurate will help to make this approach even more accurate. Another model that could probably update the generation and prediction of

sequences comes from NLP world and it is called ELMo. ELMo stands for Embeddings from Language Models and it is a language model to help predict even better a missing word from a sentence based on the appearance probability of this word in previous sentences. (Matthew E. Peters, 2018) In the problem we faced it would help understanding strong links between sequences, making prediction results more precise.

References

Brownlee, J. (2019, August 25). *Machine Leaning Mastery*. Retrieved from https://machinelearningmastery.com/models-sequence-prediction-recurrent-neural-networks/.

Brownlee, J. (2020, August 17). *Machine Learning Mastery*. Retrieved from https://machinelearningmastery.com/one-hot-encoding-for-categorical-data/

Daniel Gibert, C. M. (2020). The rise of machine learning for detection and classification malware: Research development, trends and challenges. *Journal of Network and Computer Applications*.

Eslam Amer, I. Z. (2020). A dynamic Windows malware detection and prediction method based on contextual understanding of API call sequence. *Elsevier*.

Jason, B. (2020). *Machine Learning Mastery*. Retrieved from https://machinelearningmastery.com/why-one-hot-encode-data-in-machine-learning/

Keras. (2020). *Keras.io*. Retrieved from https://keras.io

KerasDevTeam. (n.d.). *Keras.io*. Retrieved from https://keras.io/guides/sequential_model/

Mahmoud Kalash, M. R. (2018). Malware Classification with Deep Convolutional Neural Networks.

Matthew E. Peters, M. N. (2018). Deep contextualized word representations.

Olah, C. (2015). Retrieved from https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

Ralf C. Staudemeyer, E. R. (2019). Understanding LSTM a tutorial into Long Short-Term Memory Recurrent Neural Networks.

Shun Tobiyama, Y. Y. (2016). Malware Detection with Deep Neural Network. *IEEE*.

Vasudev. (2017). *Hackernoon*. Retrieved from https://hackernoon.com/what-is-one-hot-encoding-why-and-when-do-you-have-to-use-it-e3c6186d008f

ΠΑΡΑΡΤΗΜΑ

```
import numpy as np
import pandas as pd
from difflib import SequenceMatcher
import csv
import keras
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense


def load_data():
    data_path_01 = 'E:\\GeorgeChatz\\Desktop\\01.csv'
    data_path_02 = 'E:\\GeorgeChatz\\Desktop\\02.csv'
    data_path_03 = 'E:\\GeorgeChatz\\Desktop\\03.csv'
    data_path_04 = 'E:\\GeorgeChatz\\Desktop\\04.csv'
    data_path_05 = 'E:\\GeorgeChatz\\Desktop\\05.csv'
    data_path_06 = 'E:\\GeorgeChatz\\Desktop\\06.csv'
    data_path_07 = 'E:\\GeorgeChatz\\Desktop\\07.xlsx'
    datapath8 = 'E:\\GeorgeChatz\\Desktop\\08.xlsx'
    datapath8_2 = 'E:\\GeorgeChatz\\Desktop\\8.2.xlsx'

    df8_2 = pd.read_excel(datapath8_2, header=None)
    df8_2 = df8_2.iloc[::-1].reset_index(drop=True)

    df8 = pd.read_excel(datapath8, header=None)
    df8 = df8.iloc[::-1].reset_index(drop=True)

    df7 = pd.read_excel(data_path_07, header=None)
    df7 = df7.iloc[::-1].reset_index(drop=True)

    data = []
    f_array = np.array([])
    f_arr1 = np.asarray(df8_2)
    f_arr2 = np.asarray(df8)
    f_arr3 = np.asarray(df7)
    f_array = np.append(f_arr1, f_arr2, axis=0)
    f_array = np.append(f_array, f_arr3, axis=0)

    for array in f_array:
        s = np.zeros(80, dtype=int)
        for i in array:
            s[i - 1] = 1

        data.append(s)

    arr = np.asarray(data).reshape(f_array.shape[0], 1, 80)

    return arr


def decoder(array):
    decoded_array = []
    array = np.asarray(array).reshape(1, 80)
    array = array.flatten()
    array = np.asarray(array)
    bool = (array == 1)
    d_a = np.where(bool)
    decoded_array = np.array(d_a) + 1
```

```
    return decoded_array


def model(dataX, dataY, t_X, t_Y):
    """

    :type dataX: Includes the input data
    :type dataY: Includes target values

    """
    model = Sequential()
    # First model layer with MLP
    model.add(Dense(512, input_shape=(20, 80), activation='relu'))
    # feeds output to LSTM
    # LSTM input values (rows,columns,features)
    model.add(LSTM(1024, input_shape=(20, 80),
return_sequences=True))

    # Final layer with specific output/ trying to predict the next
incoming sequence
    model.add(Dense(80, activation='sigmoid'))

    model.compile(loss='binary_crossentropy',
optimizer=keras.optimizers.Adam(lr=0.0001))
    model.summary()

    model.fit(dataX, dataY, batch_size=80, epochs=640)

    for i in range(len(t_X)):
        testX, testy = t_X[i], t_Y[i]
        testX = testX.reshape(1, 1, 80)
        yhat = model.predict(testX)

        prediction = np.asarray(yhat).reshape(1, yhat.shape[2])
        prediction = prediction.flatten()
        index = np.argwhere(prediction > 0.5)
        index = index.flatten()
        index = np.array(index) + 1

        print('>Predicted', index)
        decoded_e = decoder(testy)
        decoded_e = decoded_e.flatten()
        print('>Expected=', decoded_e)
        yhat.reshape(1, 80)
        x = np.squeeze(yhat)
        decoded_p = decoder(yhat)
        print('>Predicted=', decoded_p)

        print(SequenceMatcher(None, decoded_e, index).ratio())

        with open('expected.csv', 'a', newline='') as file:
            write = csv.writer(file)
            write.writerow(decoded_e)

        with open('predicted.csv', 'a', newline='') as file:
            write = csv.writer(file)
            write.writerow(index)


def create_dataset(data, look_back=1):
    dataX, dataY = [], []
```

```
    for i in range(len(data) - look_back):
        a = data[i:(i + look_back), 0]
        b = data[i + look_back, 0]
        b = np.reshape(b, (1, 80))
        dataX.append(a)
        dataY.append(b)
    return np.array(dataX), np.array(dataY)


def similar(a, b):
    return SequenceMatcher(None, a, b).ratio()


def evaluation():
    file1 = open("expected.csv").readlines()
    file2 = open("predicted.csv").readlines()
    file1_line = []
    file2_line = []

    for lines in file1:
        file1_line.append(lines)
    for lines in file2:
        file2_line.append(lines)

    n = 0
    if (len(file1) > len(file2)) or (len(file1) < len(file2)):
        print("Error: Different file lengths")
    else:
        n = 0
        for line in file1_line:
            expected = line.split(',')
            predicted = file2_line[n].split(',')

            for i in range(0, len(expected)):
                expected[i] = int(expected[i])

            for i in range(0, len(predicted)):
                predicted[i] = int(round(float(predicted[i]), 0))

            ratio = similar(expected, predicted)

            if (ratio > 0.2):
                print("Line ", n + 1, " : ", ratio)
            n += 1


def test(x):
    print(x)
    print(decoder(x))


if __name__ == '__main__':
    data = load_data()
    t1, t2 = [], []
    print(data)
    dataX, dataY = create_dataset(data, 1)
    print(dataX.shape)
    print(dataX)

    test(dataX[0])
```

```
    for i in range(13400, 13605):
        a2 = dataX[i, 0]
        b2 = dataY[i, 0]
        t1.append(a2)
        t2.append(b2)

    testX = np.array(t1)
    testY = np.array(t2)

    dataX = np.delete(dataX, np.s_[13400:], axis=0)
    dataY = np.delete(dataY, np.s_[13400:], axis=0)
    print(dataX)
    print(testY.shape)

    model(dataX, dataY, t1, t2)
    evaluation()
```