# Análise de comportamento de malware utilizando redes neurais recorrentes - uma abordagem por intermédio da previsão de opcodes

**Davi Gomes de Albuquerque, Lincoln de Queiroz Vieira, Ricardo Sant´Ana* e Julio Cesar Duarte**
**Instituto Militar de Engenharia (IME)**
**Praça General Tibúrcio, 80, 22290-270, Praia Vermelha, Rio de Janeiro, RJ, Brasil.**
***ricardo.santana@ime.eb.br**

*RESUMO: A internet é palco de milhares de ataques cibernéticos todos os dias. Uma das maiores ameaças que afetam empresas e até usuários comuns são malware. Companhias de antivírus tentam aumentar a eficácia dos métodos de detecção de vírus, mas o grande aumento do número de variações de malware com o uso de técnicas como ofuscação aumenta a cada dia, tornando seu trabalho cada vez mais complexo. A utilização de redes neurais tem se mostrado cada vez mais presente na construção de algoritmos decisórios, inclusive na definição de classificadores de malware. Esse trabalho tem o objetivo de aplicar redes neurais recorrentes para prever os opcodes de um malware e, em seguida, classificá-los. Essa abordagem é inovadora pois, nos trabalhos analisados, não encontramos uma solução que utilize a predição de opcodes como entrada para um classificador de família de malware. O classificador de famílias obteve uma acurácia média de 92%.*

*PALAVRAS-CHAVE: Classificação. Malware. Predição. Opcodes. Redes Neurais Recorrentes. Long-Short Term Memory*

*ABSTRACT: The internet is the scene of thousands of cyber attacks every day. One of the biggest threats affecting businesses and even ordinary users is malware. Antivirus companies are trying to increase the effectiveness of virus detection methods, but the huge increase in the number of malware variations using techniques such as obfuscation is increasing every day, making their work much more complex. The use of neural networks has been shown to be increasingly present in the development of decision algorithms, including in the definition of malware classifiers. This work aims to apply recurrent neural networks to predict the opcodes of a malware and then classify them. This approach is innovative because, in the studies analyzed, we did not find a solution that uses the prediction of opcodes as input to a classifier of the malware family. The family classifier obtained an average accuracy of 92%*

*KEYWORDS: Classificação. Malware. Predição. Opcodes. Redes Neurais Recorrentes. Long-Short Term Memory*

## 1. Introdução

Apesar do desenvolvimento significativo de segurança da informação, o número de programas maliciosos, chamados de *malware*, vêm crescendo de forma espantosa a cada ano. Segundo o relatório da empresa *McAfee*, quase 150 milhões de novos *malware* foram criados durante os 8 primeiros meses de 2018 [1].

*Malware* são software projetados com intenções maliciosas que podem ser usados para espionagem, extorsão, sabotagem e para executar tarefas não autorizadas [2]. *Malware* podem, ainda, ser dos mais variados tipos como *worms*, *trojans*, *rootkits*, *spyware*, *adware*[3] e pertencerem a diversas famílias.

O método clássico para detecção de vírus utiliza um banco de dados de assinaturas digitais de *malware* que realiza a busca por padrões de código malicioso em arquivos. Porém, técnicas que alteram levemente o código malicioso burlam as assinaturas digitais dos antivírus com relativa facilidade. A criação de métodos para detectar

*malware* através de seu comportamento, por outro lado, é bastante onerosa [4].

O processamento de grande quantidade de informação relacionada a *malware* exige um esforço enorme das companhias de antivírus. Aliado a isso, existe o aumento do número de famílias de *malware* e de novas variantes dentro da família [5].

Assim, o trabalho de um analista, devido a essa evolução de códigos maliciosos, tem ficado cada vez mais complexo, cansativo e custoso. É fundamental, portanto, a automatização de tarefas dentro da análise, classificação e detecção do *malware*.

O presente trabalho teve a finalidade de elaborar, implementar e testar um algoritmo para detecção de famílias de *malware* utilizando redes neurais recorrentes. Mais especificamente, foram utilizadas dois tipos de redes neurais: uma *Long Short Term Memory* (LSTM) para realizar a predição de *opcodes*, onde se utilizou um modelo para cada família e, em seguida, uma rede neural artificial (RNA) para classificar as amostras em famílias. A partir dos modelos LSTM e RNA treinados, os *opcodes* extraídos de uma amostra de *malware* são utilizados como entrada para cada um dos modelos de LSTM. As acurácias de predição de *opcodes* de cada modelo LSTM serve de entrada para a RNA que produzirá uma predição de família da amostra.

Este artigo foi estruturado da seguinte forma: a Seção 2 descreve os principais básicos necessários ao entendimento do artigo. A Seção 3 apresenta a implementação desenvolvida e os principais resultados obtidos. A seção 4, apresenta uma breve discussão e conclusão do trabalho, além de sugestões para trabalhos futuros.

## 2. Referencial teórico

Nessa seção, serão apresentados os conceitos relacionados a *malware* e análise de *malware*, uma introdução sobre redes neurais recorrentes e LSTM, além das definições dos principais hiperparâmetros do modelo LSTM.

### 2.1 Malware e análise de malware

*Malware* é um *software* criado para ter efeitos indesejados ou danosos a um computador e oferecer grande ameaça para a segurança de sistemas computacionais [6]. Esses códigos maliciosos, muitas vezes, exploram vulnerabilidades presentes em programas, como problemas de programação na lógica de um código ou má administração do uso de memória [7].

Os vários tipos de estruturas e comportamentos de códigos maliciosos permitem agrupá-los em várias categorias. De acordo com [8], os *malware* podem ser classificados em três gerações baseando-se em três características do código malicioso: *payload*, método de propagação e utilização das vulnerabilidades. Outra classificação, baseada no comportamento do código malicioso, é proposta por [9], que divide os *malware* em *worm*, *backdoors*, *botnet*, *downloader*, *information-stealing malware*, *launcher*, *rootkit*, *scareware* e *spam-sending malware*.

Além disso, devido a evolução das técnicas de detecção de *malware* com antivírus, os autores de *malware* tem criado códigos cada vez mais complexos. A técnica mais utilizada hoje por antivírus é a de assinatura digital, em que um *malware* é identificado quando apresenta, em seu código, trechos considerados maliciosos ou quando o resultado de uma função de *hash* do

executável encontra-se numa tabela pré-definida. Essa evolução dos *malware* permite uma nova classificação: *Malware* polimórficos e *malware* metamórficos.

Malware polimórficos [10] e *malware* metamórficos [11] são capazes de alterar seus códigos através da troca da ordem de execução de instruções, introdução de instruções desnecessárias, transposição de código, substituição de instruções equivalentes entre outras técnicas. Duas são as consequências do uso dessas técnicas: a detecção desses *malware* pelos antivírus torna-se mais difícil e há um incremento no crescimento do número de variações de *malware*.

A análise de *malware* é a tarefa de examinar um *malware* de forma a entender como eles funcionam, como identificá-los e como eliminá-los. Seu propósito é, geralmente, prover informações necessárias para responder a uma intrusão em uma rede, empresa ou computador específico: descobrir o que ocorreu, assegurar-se de localizar todas as máquinas e arquivos infectados. Nessa tarefa, é importante determinar exatamente o que um artefato pode fazer, como detectá-lo e como medir e conter os danos causados [9]. Além da análise, a detecção e a classificação de *malware* são duas tarefas executadas por companhias de antivírus.

Durante a análise do *malware*, os executáveis são analisados para verificar se exibem comportamento malicioso. Com as informações coletadas sobre o *malware*, é possível armazenar seus dados em banco de dados para futura referência. A análise de *malware* feita por um analista pode ser dividida em duas fases: a estática e a dinâmica. De acordo com [9], pode-se subdividir as análises estática e dinâmica em básica e avançada.

As quatro fases, como mostra a **figura 1**, são realizadas em ciclo para que seja possível realimentar o sistema e recomeçar em qualquer fase da análise, utilizando as novas informações. Cada uma das fases será descrita a seguir.

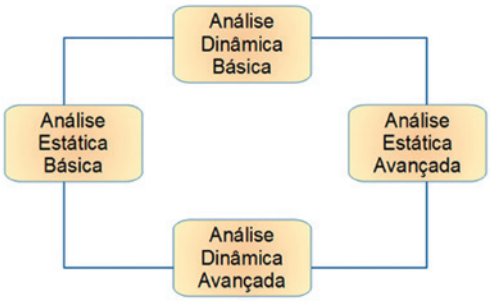


**Fig. 1 –** Fase da Análise de Malware de acordo com [9]: Análise estática básica, Análise dinâmica básica, Análise estática avançada e Análise dinâmica avançada.

Na análise estática básica, o artefato é analisado sem ser executado, procurando por padrões maliciosos em sua estrutura. Por meio desta análise, pode-se confirmar se um código é malicioso, adquirir informações básicas de suas funcionalidades e prover informações sobre possíveis conexões em redes. Por exemplo, por meio do aplicativo *Dependency Walker*, o analista é capaz de obter uma lista das funções chamadas dinamicamente pelo executável e, dessa forma, é possível prever certos comportamentos do artefato em análise. É uma fase rápida e que utiliza técnicas simples para levantar o máximo de informações do artefato mas é ineficaz contra *malware* mais sofisticados que utilizam técnicas de ofuscação, por exemplo. Os resultados obtidos dessa análise são adicionados a um relatório de análise.

A análise dinâmica básica consiste em

executar o *malware* em um ambiente virtual controlado como *sandbox* ou máquina virtual e observar seu comportamento. Com ferramentas previamente instaladas, é possível verificar quais novos processos são gerados pelo artefato, quais conexões de rede ele tenta criar, quais DLLs são chamadas e quais chaves de registro são modificadas. A medida que o analista descobre novas informações sobre o *malware*, ele atualiza o relatório com as novas informações encontradas.

A análise estática avançada é feita realizando-a engenharia reversa do código do *malware* com o auxílio de um *disassembler*. Por meio da análise das instruções *assembly* é possível descobrir o que o programa faz (sem executá-lo). Dessa forma, a análise das instruções é interessante pois pode revelar comportamentos do artefato que não foram, por exemplo, identificados durante a fase de análise dinâmica. No entanto, essa técnica é mais complexa que as demais pois exige do analista um conhecimento profundo em programação *assembly* e conceitos específicos sobre sistemas operacionais nos quais o *malware* se encontra.

A análise dinâmica avançada é realizada por meio de um *debugger* que é utilizado para examinar o estado interno de um executável (pilha, registradores) e para controlar o fluxo de chamada de cada instrução do código. Esse tipo de análise é feita quando se deseja obter informações mais precisas sobre o comportamento do artefato. Além disso, essa fase pode ser considerada complexa, uma vez que o analista deve dominar tanto o *assembly* como as principais estruturas (pilha, registradores).

Com a grande variedade e o grande crescimento do número de *malware*, as técnicas supracitadas para análise e detecção de programas maliciosos tem se tornado cada vez mais ineficazes – o tempo necessário para realizar uma análise manual é incompatível com a demanda. Já a detecção automatizada por meio de assinatura digital exige bancos de dados atualizados cada vez maiores e a criação de métodos para detectar *malware* através da semântica, por outro lado, é bastante complicada [4]. O método de análise proposto nesse trabalho se baseia na análise automatizada estática que, tem potencial para processar grande quantidade de amostras de *malware* é bem mais simples, quando comparada a análise automatizada dinâmica.

## 2.2 Redes neurais recorrentes

Redes neurais recorrentes são um tipo específico de redes neurais, onde as unidades intermediárias alimentam não apenas as camadas seguintes, mas a mesma camada ou anteriores. Ao alimentar uma célula de uma rede neural recorrente simples, é aplicada uma função de ativação, de forma que, quanto mais a informação flui pela camada intermediária, maior a quantidade de composições de funções de ativação. Ao calcular o gradiente descendente, e devido a regra da cadeia, quanto maior a sequência de composição de funções de ativação menor é o gradiente. Isso resulta em valores cada vez menores de gradiente a cada composição e, portanto, pouca ou nenhuma atualização dos pesos da rede. Esse fenômeno é chamado de *vanishing gradient*. Para solucionar esse problema foi desenvolvida a arquitetura *long short term memory* (LSTM).

## 2.3 Long Short Term Memory

Na arquitetura de uma LSTM existem 3 portas: porta de entrada, porta de esquecimento e porta

de saída. A porta de entrada é responsável por quanto o estado anterior influencia no estado atual. A porta de esquecimento controla quando parte da informação deve ser atualizada ou esquecida. A porta de saída controla quais partes da informação devem ser utilizadas na saída da célula. Cada porta é associada a pesos que regulam a atuação das mesmas [12]. Pode-se ver na **figura 2**, uma representação da LSTM. A chave para LSTMs é o estado da célula (*cell state*): a linha horizontal que percorre a parte superior do diagrama. A informação que flui pelo estado da célula sofre poucas modificações e, portanto, não sofre do efeito de *vanishing gradient*, ou seja, o gradiente não é atenuado com um número grande de estados [13].

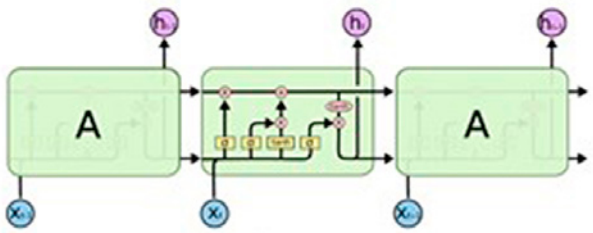


**Fig. 2 –** Camada LSTM expandida. Fonte: [14]

A lógica dentro da porta do esquecimento pode ser vista na **equação (1)**, a lógica da porta de entrada é dada pelas **equações (2) e (3)**. A composição da saída dessas duas portas no estado da célula (cell state) pode ser vista na **equação (4)**. A lógica do portão do esquecimento está apresentada em (5) e (6):

$$f_t = \sigma\,(W_f \,.\, [h_{t-1}, x_t\,] + b_f) \quad (1)$$
$$i_t = \sigma\,(W_i \,.\, [h_{t-1}, x_t\,] + b_i) \quad (2)$$
$$C_t = \tanh\,(W_C.\,[h_{t-1}, x_t\,] + b_C\,) \quad (3)$$
$$C_t = f_t * C_{t-1} + i_t * C_t \quad (4)$$
$$o_t = \sigma\,(W_o \,.\, [h_{t-1}, x_t\,] + b_o) \quad (5)$$
$$h_t = o_t * \tanh(C_t) \quad (6)$$

onde * é o produto elemento por elemento e a função σ é a função sigmoide. A **figura 3** apresenta a identificação de cada um dos portões de acordo com a nomenclatura utilizada nas equações anteriores.

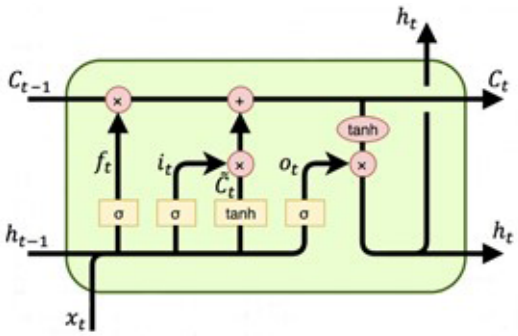


**Fig. 3 –** Célula LSTM detalhada. Baseado em [14].

Podemos dizer que $f_t$ indica o quanto da informação anterior deve ser preservado, baseado na entrada $x_t$ do estado atual e na saída $h_{t-1}$ do estado anterior. Esse termo representa a camada da porta de esquecimento. Já $i_t$ indica o que deve ser atualizado, agindo como a camada da porta de entrada, e $C_t$ elenca novas informações candidatas a serem adicionadas. Combinando $i_t$ e $C_t$ temos a atualização do estado $C_{t-1}$ para $C_t$. O termo $o_t$ define a saída que influenciará os próximos estados, caracterizando a porta de saída. Os termos $b_f$, $b_i$ e $b_c$ presentes nas equações correspondem ao *bias*, critérios de preferência de uma hipótese de sobre outra, e os pesos $W_f$, $W_i$, $W_c$ e $W_o$ presentes nas camadas da LSTM estão profundamente relacionados ao processo de aprendizado, sendo esses parâmetros adaptados com o intuito de diminuir o erro e obter um modelo eficiente [14].

## 2.4 Hiperparâmetros do modelo LSTM

Para o presente trabalho foram realizadas

escolhas de hiperparâmetros relacionadas à implementação da LSTM na biblioteca Keras. Abaixo temos uma relação dos hiperparâmetros relevantes e sua definição.

- *lookback*: indica quantas etapas anteriores devem ser importantes na predição da próxima etapa [15].
- *batch size*: indica o número de exemplos de treinos usados em uma iteração do treino. A rede neural recorrente não pode tratar todos os exemplos de treino por limitação de memória.
- *epoch*: representa uma varredura pelo conjunto inteiro do conjunto de dados de treinamento durante o treinamento do modelo. Faz-se necessário usar mais de um *epoch*, para melhor adaptar o modelo a todo conjunto de dados [16].
- *time step*: reflete o ponto de observação da amostra, ou seja, qual o tamanho temporal da sequência de amostras que vai alimentar o modelo. Mesmo valor de *lookback*.
- otimizadores: tem função de acelerar o processo de treino produzindo resultados compatíveis. O otimizador escolhido foi o *adam* por ser o mais utilizado. Existem outros otimizadores para casos mais específicos [17].

## 3. Implementação e resultados

Nesta seção serão apresentados as implementações e experimentos realizados além dos resultados. Será descrito como foi realizada a extração e codificação dos *opcodes*, além do treinamento do modelo de predição de *opcodes* e do modelo de classificação em famílias com seus respectivos resultados.

### 3.1 Extração de Opcodes

A base de dados utilizada foi obtida do Desafio de Classificação de Malware da Microsoft de 2015 [26] e contém 10.868 amostras de *malwares* classificados em 9 famílias. Cada amostra é composta por um arquivo do tipo *asm*, contendo o código em *assembly* do *malware*, e um arquivo do tipo *byte*, com um extrato do executável em binário. Os *opcodes* foram extraídos dos arquivos *asm* por meio de rotinas de processamento de texto. A **tabela 1** mostra a quantidade original de amostras por classe.

**Tab. 1** – Distribuição das famílias da malware na base de dados da Microsoft [26].

| Ordem | Nome da Família | Quantidade de Amostras |
|---|---|---|
| 1 | Ramnit | 1541 |
| 2 | Lollipop | 2478 |
| 3 | Kelihos_ver3 | 2942 |
| 4 | Vundo | 475 |
| 5 | Simda | 42 |
| 6 | Tracur | 751 |
| 7 | Kelihos_ver1 | 398 |
| 8 | Obfuscator.ACY | 1228 |
| 9 | Gatak | 1013 |

### 3.2 Codificação dos Opcodes

Para codificar os *opcodes* foi utilizada a técnica *word2vec*, que é um método para criar *embeddings* entre palavras ao transformá-las em vetores. O *word2vec*, como o nome sugere, tenta descrever palavras como forma de vetores, agrupando palavras com semânticas similares de forma mais próxima, onde cada dimensão do vetor é entendida como uma característica da palavra. Geralmente, palavras da língua inglesa, codificadas com *word2vec*, possuem dimensão 300. Ainda assim, é possível visualizar os vetores em um

gráfico duas dimensões, utilizando técnicas de redução de dimensionalidade, como por exemplo a análise de componentes principais (*Principal Component Analysis* - PCA). Na **figura 4** temos nome de países e suas respectivas capitais projetadas em 2 dimensões:

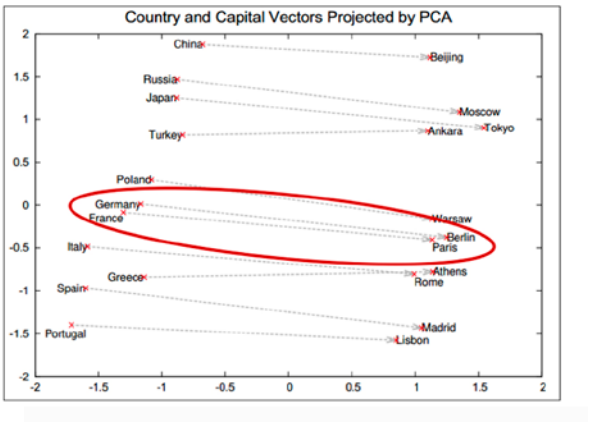


**Fig. 4 –** Vetores de exemplos com países e capitais. Fonte: [18]

Podemos observar que os vetores de diferença entre capital e país é quase constante, o que mostra que a técnica de *word2vec* foi capaz de compreender a relação entre esses dois tipos de palavras (nome do país e capital).

Neste trabalho foi utilizado uma codificação fruto da aplicação do método *word2vec* às instruções *assembly*. Esses vetores foram gerados por [19], criando uma base de dados de instruções assembly na forma de vetores de dimensão 100.

A partir dos vetores criados por [19], que consideram o *opcode* e os operandos para formar o vetor, foram criados novos vetores para representar apenas os *opcodes*. Isso foi realizado calculando a média entre todos os vetores existentes que continham o *opcode* desejado, independente do operando. Por exemplo, para calcular o vetor correspondente ao *opcode mov* foi calculada a média entre os 107.339 vetores presentes na base de dados de [19]. O resultado deste procedimento aplicado a cada um dos *opcodes* extraídos neste trabalho foi denominado de *opcode2vec*.

Para observarmos como essa codificação representa a semântica do *opcode* apresentamos na **figura 5** quatro instruções projetadas em duas dimensões: *add* (ponto vermelho), *sub* (ponto verde), *pop* (ponto amarelo) e *push* (ponto azul). É importante ressaltar que os eixos X e Y da **figura 5** não tem significado pois são eixos de projeção do vetor de dimensão 100.

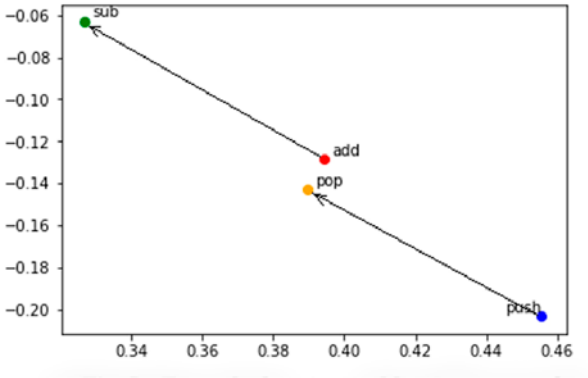


**Fig. 5 –** Exemplo de vetores add, sub, pop e push projetados no plano e utilizados neste trabalho.

Da **figura 5**, podemos observar que a diferença *sub* - *add* é aproximadamente igual à diferença *pop* – *add*. Desta forma, podemos concluir que o vetor diferença representa função oposta. Essa codificação de *opcodes* foi utilizada no conjunto de dados de treinamento e testes. A **figura 6** apresenta a sequência *push,push,call,jnz,mov, …, push, int* e *xor* sendo codificada para vetores. Podemos observar que *opcodes* iguais são codificados para vetores iguais.

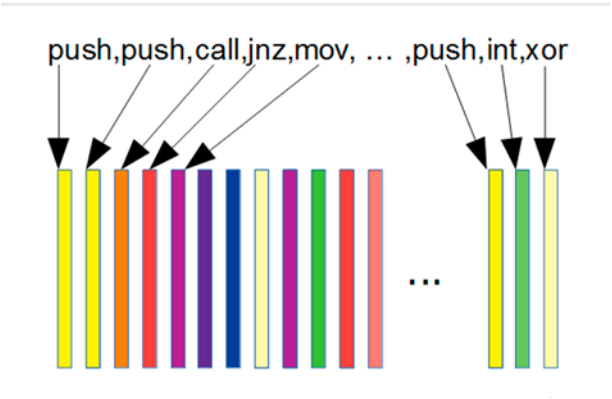


**Fig. 6 –** Processo de codificação de uma sequência de opcodes em vetores utilizando opcodes2vec.

Para alimentar o modelo LSTM, o tamanho da sequência temporal (*lookback*) foi definida como 5, ou seja, sequências de entrada de 5 *opcodes* consecutivos (entrada) para predizer o sexto elemento, a saída desejada.

Exemplificando, um *malware* que contém a sequência de *opcodes*: op1 (tempo 1), op2 (tempo 2), op3 (tempo 3), op4 (tempo 4), op5 (tempo 5), op6 (tempo 6) tem como entrada (op1, op2, op3, op4, op5) e como rótulo de saída op6.

A **figura 7** apresenta um exemplo onde, a partir de uma sequência de *opcodes* codificados foram geradas 3 sequências contendo a entrada de tamanho 5 e a saída desejada (sexto elemento da sequência).

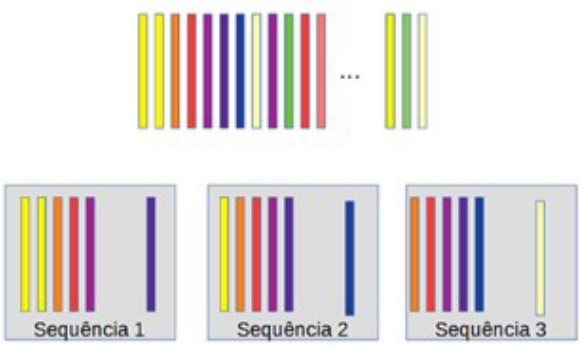


**Fig. 7 –** Sequência de vetores codificados gerando as 3 primeiras sequências com loopback igual a 5.

## 3.3 Treinamento dos modelos de predição de *Opcodes*

As entradas das sequências de *opcodes* codificadas de cada classes e a saída desejada servem para treinar a LSTM. A **figura 8** apresenta uma sequência de vetores codificados (entrada) e a saída desejada (rótulo) utilizados para treinar a rede LSTM para predição *opcodes*. Para gerar um modelo LSTM para cada classe, esse processo é repetido para cada sequência de entrada e saída desejada das amostras daquela classe. Ao final, teremos 9 modelos preditores de *opcodes*: um para cada família de *malware*.

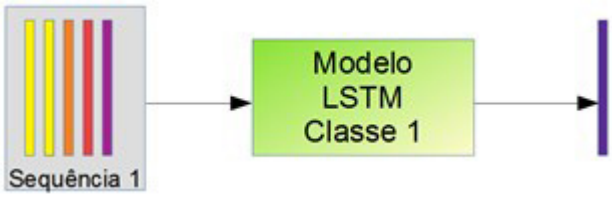


**Fig. 8 –** Cada sequência de vetores codificados de tamanho 5 o rótulo de saída são utilizados para treinar a rede LSTM.

A dimensão da LSTM é a dimensão do vetor de saída e o *time step* do modelo é igual ao valor do *lookback*, ou seja, 5. O treinamento foi feito minimizando o erro quadrático médio, até a convergência do erro, com *batch size* = 100 por 100 épocas.

A taxa de acertos da predição de *opcodes* para o conjunto de testes, utilizando elementos da própria família, é apresentado na **tabela 2** dentro das respectivas classes.

**Tab. 2 –** Avaliação do treino dos modelos de predição de opcodes usando opcode2vec

| Nome da Família | Acurácia de Predição de opcodes |
|---|---|
| Família 1 | 40,6% |
| Família 2 | 37,9% |
| Família 3 | 73,4% |
| Família 4 | 53,6% |
| Família 5 | 94,0% |
| Família 6 | 91,3% |
| Família 7 | 69,1% |
| Família 8 | 80,3% |
| Família 9 | 90,1% |

Da tabela, podemos concluir que os modelos LSTM foram capazes de modelar as amostras de treinamento de cada uma das classes. As classes 1, 2 e 4 ficaram com as menores acurácias na predição de *opcodes*: provavelmente o modelo LSTM utilizado não foi capaz de modelar a diversas sequências encontradas nesta classe. Ainda assim, isso não é um fator limitante pois o objetivo principal é que o modelo da classe deva ser capaz de fazer predições da própria classe melhor que os outros modelos.

A saída dos modelo LSTM – que é a acurácia de predição de *opcodes*, foi utilizada para treinar uma RNA.

## 3.4 Treinamento da RNA para classificação

Após o treino dos modelos utilizando LSTM, cada um dos nove modelos realizou predições sobre cada uma das amostras das nove famílias e as acurácias foram obtidas. A **figura 9** apresenta as sequências de uma determinada amostra alimentando os 9 modelos preditores de *opcodes* e produzindo um vetor de acurácias de dimensão 9.

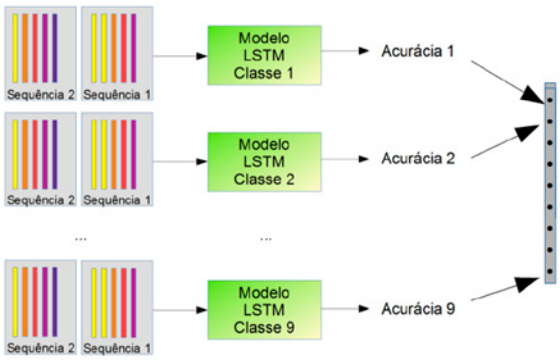


**Fig. 9** – As sequências de opcodes de uma determinada amostra alimentam os 9 modelos preditores de opcodes. Um vetor de acurácias de dimensão 9 é produzido. Na figura apresentamos apenas as duas primeiras sequências de uma determinada amostra.

Com os resultados das predições (vetor de acurácias) de cada amostra de *malware*, foi possível treinar uma rede neural artificial para classificar cada amostra em famílias. A rede neural utilizada possui duas camadas sendo que a primeira camada com 9 neurônios, duas camadas escondidas com 12 e 10 neurônios respectivamente e a camada de saída com 9 neurônios. A **figura 10** apresenta um vetor de acurácias (para as diversas amostras de *malware*) utilizados para treinamento de rede neural artificial.

Na última camada foi utilizada uma função de ativação chamada de *softmax*, que força a soma de todas as saídas ser igual a 1, permitindo uma interpretação probabilística de cada uma das 9 saídas da rede neural.

Para o treinamento da rede neural artificial, foram utilizadas 80% do conjunto total de amostras de treinamento, totalizando 8604 amostras. A entrada da rede neural artificial é um vetor de 9 posições: a acurácia de predição de *opcodes* para cada uma das 9 famílias.

Dessa forma, a rede neural artificial aprendeu, portanto, as relações entre as acurácias de forma a classificar corretamente os *malware* em famílias. Para o teste da rede neural artificial, todo o conjunto de teste foi utilizado.

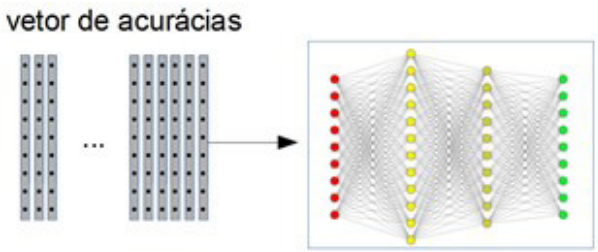


**Fig. 10** – As sequências de opcodes de uma determinada amostra alimentam os 9 modelos preditores de opcodes. Um vetor de acurácias de dimensão 9 é produzido. Na figura apresentamos apenas as duas primeiras sequências de uma determinada amostra.

Para definir os melhores hiperparâmetros da rede neural artificial, foram realizados experimentos com diversos valores de hiperparâmetros:

- batch size: 2,5, 10, 20;
- epochs: 500, 1000, 1500;

Função de perda: foram testadas as funções binary cross entropy e mean squared error.

A rede neural artificial obteve melhores resultados com os seguintes hiperparâmetros:

- batch size: 20;
- epochs: 1000;
- Função de perda: m*ean squared error*.

A acurácia do modelo para o conjunto de dados de teste foi de 92%. A **tabela 3** mostra os resultados da rede neural artificial utilizando outras métricas para cada uma das famílias.

Considerando que a distribuição de amostras entre as famílias não é balanceada, consideramos utilizar o *f1-score*. O *f1-score* é uma medida de desempenho comumente utilizada para classificações com múltiplos rótulos [21]. É calculada com a média harmônica dos valores abrangência e precisão, ou seja, ela busca um equilíbrio entre esses valores [22], penalizando-o pelo menor dos valores. De acordo com [22], valores altos de precisão e abrangência significam que a classe está bem definida no modelo.

Dessa forma, as famílias mais bem definidas no modelo são as famílias 3,4 e 8, pois seus valores de *f1-score* são os mais altos. Com exceção da família 5, todas as famílias produziram um *f1-score* acima de 0.80.

**Tab. 3** – Resultados do classificador

| Nome da Família | Precisão | Abrangência | f1-score |
|---|---|---|---|
| Família 1 | 0,88 | 0,92 | 0,90 |
| Família 2 | 0,84 | 0,97 | 0,90 |
| Família 3 | 1,00 | 1,00 | 1,00 |
| Família 4 | 0,97 | 0,96 | 0,97 |
| Família 5 | 1,00 | 0,21 | 0,35 |
| Família 6 | 0,94 | 0,87 | 0,91 |
| Família 7 | 1,00 | 0,83 | 0,91 |
| Família 8 | 0,97 | 0,86 | 0,92 |
| Família 9 | 0,91 | 0,74 | 0,81 |

Possivelmente, o valor baixo de *f1-score* para a família 5 se justifique pela pequena quantidade de amostras disponíveis dessa classe. O modelo LSTM preditor de *opcodes* da classe 5 obteve uma acurácia de 94% - no entanto, esse modelo não deve ser capaz distinguir amostras da classe 5 de outras, produzindo um resultado ruim para o *f1-score* de classificação da rede neural artificial.

Para comparação de resultados, Rocca [23] é referência principal. Esse autor realiza a extração de *opcodes* da mesma base de dados da *Microsoft* utilizada neste trabalho. Em seguida, implementa uma codificação de bigramas de *opcodes* utilizando o *Term Frequency - Inverse Document Frequency* - TF-IDF [24]. O TF-IDF tenta indicar a importância de uma palavra (*opcode*) de um binário em relação a uma coleção de binários.

A classificação em famílias é feita por meio de uma rede auto-codificadora pré-treinada para reconstruir as entradas TF-IDF. A saída da metade codificadora dessa rede é usada como entrada para uma rede neural artificial para classificação em famílias. Desta forma, pode-se observar que essa solução possui as mesmas limitações de classificação de um artefato em uma família deste trabalho, ao utilizar como informação para classificação apenas os *opcodes* das amostras. Desta forma, os resultados passam a ser comparáveis com o que foi desenvolvido neste trabalho.

A **tabela 4** apresenta os resultados obtidos pelo classificador [23]. Podemos observar que os valores de *f1-score* foram, no geral, superiores aos alcançados neste trabalho. Pode-se observar, no entanto, algumas semelhanças: a família 5 obteve, em ambos os casos, um valor de *f1-score* mais baixo, possivelmente por causa da pouca quantidade de amostras da família 5. Além disso, o maior *f1-score* foi obtido em ambos os casos na família 3, pois essa

possui a maior quantidade de amostras.

**Tab. 4** – Métricas por classe para o classificador final – Rede 4, 4 camadas ocultas, conjunto de dados de bigramas. Fonte: [23]

| Nome da Família | Precisão | Abrangência | f1-score |
|---|---|---|---|
| Família 1 | 0,97 | 0,94 | 0,96 |
| Família 2 | 0,97 | 0,98 | 0,97 |
| Família 3 | 0,99 | 0,99 | 0,99 |
| Família 4 | 0,96 | 0,89 | 0,93 |
| Família 5 | 0,72 | 0,72 | 0,72 |
| Família 6 | 0,89 | 0,96 | 0,92 |
| Família 7 | 0,97 | 0,95 | 0,96 |
| Família 8 | 0,92 | 0,95 | 0,93 |
| Família 9 | 0,97 | 0,94 | 0,96 |

Podemos observar que, em sua grande maioria, o modelo proposto obteve melhores resultados de f*1-score* que os apresentados neste trabalho [23]. De qualquer forma, devido a limitação de tempo, não tivemos oportunidade de avaliar o impacto do ajuste de diversos hiperparâmetros no *f1-score* da classificação de cada família. Essas sugestões de melhoria serão apresentadas na seção seguinte na forma de trabalhos futuros.

## 4. Discussão e conclusão

A criação de modelos preditivos de *opcodes* para cada uma das 9 famílias de *malware* permite a implementação de um classificador em famílias baseado nas acurácias dos modelos preditores de forma automatizada, motivados pela dificuldade em classificar um grande volume de artefatos.

Uma dificuldade inesperada foi encontrada na extração dos *opcodes*, pois foi necessário o desenvolvimento de um *script* especializado em extrair *opcodes* dos textos exportados pelo IDA – seria mais interessante utilizar bibliotecas especializadas para essa extração, mas para isso seria necessário possuir o binário do artefato malicioso. Tão importante quanto a correta extração de opcodes foi a correta codificação dos opcodes. Neste trabalho, os resultados mais interessantes foram obtidos utilizando uma codificação de opcodes baseadas no word2vec utilizando uma base de codificação desenvolvida [19].

Foram montados 9 modelos preditores de opcodes usando rede neural recorrente baseada em LSTM: consideramos que os resultados obtidos na predição de opcodes pelos 9 modelos foi adequada para desenvolver um classificador de família baseado na saída dos preditores de opcodes.

Os resultados apresentados pelo classificador foram satisfatórios, apesar do resultado ruim para a família 5. Além disso, os resultados obtidos em [23], que também utiliza a informação de extração de opcodes, foram superiores aos apresentados nestes trabalhos, motivando a sugestão de trabalhos futuros que possam melhorar os resultados aqui apresentados.

Assim, como trabalhos futuros, pode-se sugerir:

- Extrair opcodes e operandos, pois dessa forma são obtidas mais informações para o modelo de predição. Para isso, seria necessário gerar uma base mais completa de codificação baseada no word2vec que codifique opcodes e operandos juntos.
- Aplicar as técnicas aqui apresentadas em outra base de dados de malware, que não estivesse limitada a 10.000 amostras.
- Utilizar outras codificações de opcodes/operandos como fastText [20] ou BERT [21].
- Ajustar alguns hiperparâmetros como: o time step para predição, o número de épocas de forma a garantir a convergência do erro (por exemplo, por meio do parâmetro early stop).
- Utilizar uma estrutura LSTM mais complexa, como LSTM empilhadas, sempre levando em conta a possibilidade de overfitting e maior duração dos treinamentos.

## Referências bibliográficas


[1] [1] RAJ SAMANI, C. B. McAfee Labs Threads Report. Disponível em: <https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-dec-2018.pdf>. Acesso em: 31 dec. de 2018.

[2] [2] RAYMOND J. CANZANESE, J. Detection and Classification of Malicious Processes Using System Call Analysis. 2015. 1 f. Tese (Doctor in Philosophy) – Drexel University, Filadélfia, Estados Unidos, 2015.

[3] [3] GAVRILUT, D.; CIMPOESU, M.; ANTON, D. ; CIORTUZ, L. Malware detection using machine learning. Proceedings of the International Multiconference on Computer Science and Information Technology, v. 4, p. 735–741, 2009.

[4] [4] UCCI, D.; ANIELLO, L. ; BALDONI, R. Survey on the usage of machine learning techniques for malware analysis. Computers & Security, v. 81, p. 123–147, 2017.

[5] [5] AHMADI, M.; ULYANOV, D.; SEMENOV, S.; TROFIMOV, M. ; GIACINTO, G. Novel feature extraction, selection and fusion for effective malware family classification. In: CODASPY, 16., 2016. Anais... [S.l.: s.n.], 2016, p. 183–194.

[6] [6] YUXIN, D.; SIYI, Z. Malware detection based on deep learning algorithm. Neural Computing and Applications, v. 31, n. 2, p. 461–472, 2019.

[7] [7] KUMAR, P. D.; NEMA, A. ; KUMAR, R. Hybrid analysis of executables to detect security vulnerabilities: security vulnerabilities. In: PROCEEDINGS OF THE 2ND INDIA SOFTWARE ENGINEERING CONFERENCE, 2., 2009. Anais... [S.l.: s.n.], 2009, p. 141–142.

[8] [8] SUNG, A. H.; MUKKAMALA, S. ; XU, J. Static analyzer of vicius executables. Proceedings - Annual Computer Security Applications Conference, ACSAC, v. 0, p. 326–334, 2005.

[9] [9] SIKORSKI, M.; HOGIN, A. Pratical Malware Analysis. 5. ed. [S.l.]: No Starch Press, 2012. 2-2 p.

[10] [10] MATHUR, K.; HIRANWAL, S. A survey on techniques in detection and analyzing malware executable. International Journal of Advanced Research in Computer Science and Software Engineering, v. 3, p. 422–428, 2013.

[11] [11] CHRISTODORSCU, M.; JHA, S. Static analysis of executables to detect malicious patterns. In: USENIX SECURITY SYMPOSIUM, 12., 2004. Anais... [S.l.: s.n.], 2004, p. 169–186.

[12] [12] JONES, M. T. Arquiteturas de aprendizado profundo. Disponível em: <https://www.ibm.com/developerworks/br/library/cc-machine-learning-deeplearning-architectures/cc-machine-learning-deep-learning-architectures-pdf.pdf>.Acesso em: 10 mai. de 2019.

[13] [13] BUDUMA, N.; LOCASCIO, N. Fundamentals of Deep Learning: Designing Next-Generation Machine Intelligence Algorithms. 1st. ed. [S.l.]: O'Reilly Media, Inc., 2017. ISBN 1491925612, 9781491925614.

[14] [14] OLAH, C. Understanding LSTM Networks. Disponível em: <https://colah.github.io/posts/2015-08-Understanding-LSTMs/>. Acesso em: 10 mai. de 2019.

[15] [15] KOMPELLA, R. Using LSTMs to forecast time-series. Disponível em: <https://towardsdatascience.com/using-lstms-to-forecast-time-series--4ab688386b1f>. Acesso em: 10 mai. de 2019.

[16] [16] SHARMA, S. Epoch vs Batch Size vs Iterations. Disponível em: <https://towardsdatascience.com/epoch-vs-iterations-vs-batch-size-4dfb9c7ce9c9>. Acesso em: 10 mai. de 2019.

[17] [17] BROWNLEE, J. Gentle Introduction to the Adam Optimization Algorithm for Deep Learning. Disponível em: https://machinelearningmastery.com/adam--optimizationalgorithm-for-deep-learning/>. Acesso em: 10 mai. de 2019.

[18] [18] GOM, J. Mining English and Korean text with Python. Disponível em: <https://www.stechstar.com/user/zbxe/AlgorithmPython/52575>. Acesso em: 01 outubro de 2019.

[19] [19] MASSARELLI, L.; DI LUNA, G. A.; PETRONI, F.; BALDONI, R. ; QUERZONI, L. Safe: Self-attentive function embeddings for binary similarity. In: INTERNATIONAL CONFERENCE ON DETECTION OF INTRUSIONS AND MALWARE, AND VULNERABILITY ASSESSMENT, 16., 2019. Anais... [S.l.: s.n.], 2019, p. 309–329.

[20] [20] DI, W.; BHARDWAJ, A. ; WEI, J. Deep Learning Essentials: Your Hands-on Guide to the Fundamentals of Deep Learning and Neural Network Modeling. [S.l.]: Packt Publishing, 2018. ISBN 1785880365, 9781785880360.

[21] [21] DEVLIN, J.; CHANG, M.; LEE, K. ; TOUTANOVA, K. BERT: pre-training of deep bidirectional transformers for language understanding. CoRR, v. abs/1810.04805, 2018. Disponível em: <http://arxiv.org/abs/1810.04805>. Acesso em: 3 out. de 2019.

[22] [21] FUJINO, A.; ISOZAKI, H. ; SUZUKI, J. Multi-label text categorization with model combination based on f1-score maximization. In: PROCEEDINGS OF THE THIRD INTERNATIONAL JOINT CONFERENCE ON NATURAL LANGUAGE PROCESSING: VOLUME-II, 3., 2008. Acesso em: 3 out. de 2019

[23] [22] BAPTISTE ROCCA. Handling imbalanced datasets in machine learning. Disponível em: <https://towardsdatascience.com/handling-imbalanced -datasets-in-machine-learning-7a0e84220f28>. Acesso em: 27 de janeiro de 2019.

[24] [23] PINTO, D. R. Aprendizado Profundo Aplicado à Análise Estática de Malwares. Disponível em: <http://www.comp.ime.eb.br/pos/arquivos/publicacoes/dissertacoes/2018/2018-Dhiego.pdf>. Acesso em: 107 fev. de 2020.

[25] [24] MANNING, C. D.; SCHÜTZE, H. ; RAGHAVAN, P. Introduction to information retrieval. New York, NY, USA: Cambridge University Press, 2008.

[26] [25] KAGGLE, T. Microsoft Malware Winners' Interview: 1st place, "NO to overfitting!". Disponível em: <http://blog.kaggle.com/2015/05/26/microsoft-malwarewinners-interview-1st-place-no-to-overfitting/>. Acesso em: 29 setembro de 2019.