

**MINISTÉRIO DA DEFESA
EXÉRCITO BRASILEIRO
DEPARTAMENTO DE CIÊNCIA E TECNOLOGIA
INSTITUTO MILITAR DE ENGENHARIA
CURSO DE GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO**

**1º Ten DAVI GOMES DE ALBUQUERQUE
1º Ten LINCOLN DE QUEIROZ VIEIRA**

**CLASSIFICAÇÃO DE MALWARE UTILIZANDO ANÁLISE ESTÁTICA E
REDES NEURAIS RECORRENTES**

**Rio de Janeiro
2019**

INSTITUTO MILITAR DE ENGENHARIA

1º Ten DAVI GOMES DE ALBUQUERQUE
1º Ten LINCOLN DE QUEIROZ VIEIRA

**CLASSIFICAÇÃO DE MALWARE UTILIZANDO ANÁLISE
ESTÁTICA E REDES NEURAIIS RECORRENTES**

Projeto de Fim de Curso apresentado ao Curso de Graduação em Engenharia de Computação do Instituto Militar de Engenharia, como requisito parcial para a obtenção do título de Engenheiro de Computação.

Orientador: TC Julio Cesar Duarte - D.Sc.

Co-Orientador: TC Ricardo Sant'Ana - M.Sc.

Rio de Janeiro
2019

INSTITUTO MILITAR DE ENGENHARIA
Praça General Tibúrcio, 80 - Praia Vermelha
Rio de Janeiro - RJ CEP 22290-270

Este exemplar é de propriedade do Instituto Militar de Engenharia, que poderá incluí-lo em base de dados, armazenar em computador, microfilmar ou adotar qualquer forma de arquivamento.

É permitida a menção, reprodução parcial ou integral e a transmissão entre bibliotecas deste trabalho, sem modificação de seu texto, em qualquer meio que esteja ou venha a ser fixado, para pesquisa acadêmica, comentários e citações, desde que sem finalidade comercial e que seja feita a referência bibliográfica completa.

Os conceitos expressos neste trabalho são de responsabilidade do(s) autor(es) e do(s) orientador(es).

de Albuquerque, Davi Gomes

Classificação de malware utilizando análise estática e redes neurais recorrentes / Davi Gomes de Albuquerque, Lincoln de Queiroz Vieira, orientado por Julio Cesar Duarte e Ricardo Sant'Ana - Rio de Janeiro: Instituto Militar de Engenharia, 2019.

59p.: il.

Projeto de Fim de Curso (graduação) - Instituto Militar de Engenharia, Rio de Janeiro, 2019.

1. Curso de Graduação em Engenharia de Computação - projeto de fim de curso. 1. análise de *malware*. 2. redes neurais recorrentes. 3. predição. 4. *opcode*. 5. *long short-term memory*. 6. codificação *Word2vec*. I. Duarte, Julio Cesar. II. Sant'Ana, Ricardo. III. Título. IV. Instituto Militar de Engenharia.

INSTITUTO MILITAR DE ENGENHARIA

1º Ten DAVI GOMES DE ALBUQUERQUE
1º Ten LINCOLN DE QUEIROZ VIEIRA

CLASSIFICAÇÃO DE MALWARE UTILIZANDO ANÁLISE
ESTÁTICA E REDES NEURAIS RECORRENTES

Projeto de Fim de Curso apresentado ao Curso de Graduação em Engenharia de Computação do Instituto Militar de Engenharia, como requisito parcial para a obtenção do título de Engenheiro de Computação.

Orientador: TC Julio Cesar Duarte - D.Sc.

Co-Orientador: TC Ricardo Sant'Ana - M.Sc.


Aprovado em 09 de Outubro de 2019 pela seguinte Banca Examinadora:



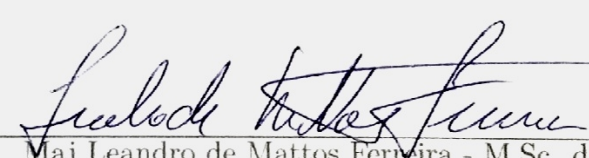
TC Julio Cesar Duarte - D.Sc. do IME - Presidente



TC Ricardo Sant'Ana - M.Sc. do IME



Prof. Raquel Coelho Gomes Pinto - D.Sc. do IME



Maj Leandro de Mattos Ferreira - M.Sc. do IME

Rio de Janeiro
2019

Ao Instituto Militar de Engenharia, alicerce da minha formação e aperfeiçoamento.

AGRADECIMENTOS

Agradeço a todas as pessoas que nos incentivaram, apoiaram e possibilitaram esta oportunidade de ampliar meus horizontes.

Nossos familiares, cônjuge e mestres.

Em especial ao nosso professor orientador Julio Cesar Duarte e ao co-orientador Ricardo Sant'Ana, por suas disponibilidades e atenções.

“Sem publicação, a ciência é morta. ”

GERARD PIEL

SUMÁRIO

LISTA DE ILUSTRAÇÕES	8
LISTA DE TABELAS	9
LISTA DE SIGLAS	10
LISTA DE ABREVIATURAS	11
1 INTRODUÇÃO	14
1.1 Motivação	14
1.2 Objetivos	15
1.3 Justificativa	16
1.4 Metodologia	16
1.5 Estrutura do texto	17
2 REFERENCIAL TEÓRICO	18
2.1 <i>Malware</i>	18
2.1.1 Tipos e Variações de <i>malware</i>	19
2.2 Análise de <i>Malware</i>	21
2.2.1 Fases da análise de <i>malware</i>	21
2.2.1.1 Análise estática básica	21
2.2.1.2 Análise dinâmica básica	22
2.2.1.3 Análise estática avançada	22
2.2.1.4 Análise dinâmica avançada	23
2.2.2 Técnicas de detecção de <i>malware</i>	23
2.2.3 Técnicas anti-análise e anti-deteção	24
2.2.3.1 <i>Anti-disassembly</i>	24
2.2.3.2 <i>Anti-debugging</i>	24
2.2.3.3 <i>Anti-virtual Machine</i>	25
2.2.3.4 Ofuscação	25
2.3 Aprendizado Profundo	26
2.3.1 Redes Neurais	27
2.3.2 <i>Backpropagation</i>	28
2.3.3 Redes Neurais Recorrentes	29

2.3.4	<i>Long short-term memory - LSTM</i>	29
2.3.4.1	Aspectos matemáticos	30
2.3.4.2	Aplicação	31
3	UMA PROPOSTA DE PREDIÇÃO DE OPCODES PARA A CLASSIFICAÇÃO DE MALWARE EM FAMÍLIAS	32
4	EXTRAÇÃO DE OPCODES	34
4.1	Extração de instruções	34
4.2	Problemas na extração	35
5	APLICAÇÃO DE ALGORITMOS DE PREDIÇÃO	39
5.1	Atividades de pré-processamento	39
5.2	Representação dos <i>opcodes</i>	39
5.3	Modelagem utilizada para os preditores	41
5.4	Definição e treinamento dos modelos	43
5.5	Resultados dos treinamentos	43
6	DEFINIÇÃO E TREINAMENTO DO MODELO DO CLASSIFICADOR DE FAMÍLIAS	46
6.1	Sistema de Votação Simples	46
6.2	Classificador com rede neural	48
7	CONCLUSÃO	53
8	REFERÊNCIAS BIBLIOGRÁFICAS	55

LISTA DE ILUSTRAÇÕES

FIG.2.1	Fases da análise de <i>malware</i>	22
FIG.2.2	Diferença entre aprendizado de máquina e aprendizado profundo (ODI; NGUYEN, 2018)	27
FIG.2.3	Descrição funcional da estrutura do neurônio (BUDUMA; LOCAS- CIO, 2017)	27
FIG.2.4	Exemplo de rede neural com três camadas com três neurônios por camada(BUDUMA; LOCASCIO, 2017)	28
FIG.2.5	Camada LSTM expandida (OLAH, 2015)	29
FIG.2.6	Célula LSTM detalhada (LI et al., 2017)	30
FIG.3.1	Diagrama da primeira parte do trabalho	32
FIG.3.2	Diagrama da segunda parte do trabalho	33
FIG.3.3	Diagrama da fase de aplicação dos modelos	33
FIG.5.1	Vetores de exemplos com cores (ALLAMAR, 2019)	40
FIG.5.2	Vetores de exemplos com países e capitais (GOM, 2015)	41
FIG.5.3	Exemplo de vetores com dimensão reduzida de <i>opcodes</i> utilizados no trabalho	42
FIG.5.4	Gráfico do erro do modelo 4 por quantidade de épocas	44
FIG.5.5	Gráfico do erro do modelo 2 por quantidade de épocas	45
FIG.6.1	Estrutura da rede neural utilizada no classificador	48

LISTA DE TABELAS

TAB.4.1	Número de amostras em cada classe	34
TAB.4.2	Número de amostras em cada classe com código em diferentes se- ções	36
TAB.4.3	Número de amostras em cada classe apenas com instruções db, dd, dw	38
TAB.5.1	Avaliação do treino dos modelos usando codificação escalar	44
TAB.5.2	Avaliação do treino dos modelos usando a codificação proveniente do Opcode2vec	45
TAB.6.1	Matriz de confusão do sistema classificador de votação simples para 1.289 amostras de teste.	46
TAB.6.2	Resultados gerais do sistema de votação	47
TAB.6.3	Matriz de confusão do sistema classificador baseado em rede neural com hiper-parâmetros <i>epochs</i> =1500, <i>batch size</i> =2 e função de perda MSE para 1.289 amostras de teste.	50
TAB.6.4	Resultados gerais do classificador utilizando rede neural	50
TAB.6.5	Métricas por classe para o classificador final – Rede 4, 4 camadas ocultas, conjunto de dados de Bigramas.(PINTO, 2018)	51
TAB.6.6	Métricas por família para a solução ganhadora do concurso <i>Micro- soft/Kaggle</i> (KAGGLE, 2015)	52

LISTA DE SIGLAS

LSTM	<i>Long Short-Term Memory</i>
IME	Instituto Militar de Engenharia
DLL	Dynamic-Link Library
RCP	Remote copy
RSH	Remote shell
SCADA	Supervisory Control and Data Acquisition
NLP	Natural Language Processing

LISTA DE ABREVIATURAS

SÍMBOLOS

σ - Função sigmoid

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*, *software* maliciosos que infectam computadores com o objetivo de modificar seu funcionamento. 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 e difícil. A utilização de redes neurais tem se mostrado cada vez mais presente na construção de algoritmos decisórios, inclusive na construção 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 91%.

ABSTRACT

The internet is the scene of thousands of cyber attacks every day. One of the biggest threats that affect companies and even common users are malware, malicious software that infect computers with the purpose of modifying their operation. Anti-virus technologies companies try to increase the effectiveness of virus detection methods, but the increase in the number of malware with the use of techniques such as obfuscation have been increasing, making their work increasingly complex and difficult. Deep neural networks utilization has been increasingly present in the algorithms construction . This work aims to apply recurring neural networks to predict the opcodes of a malware and then to be able to classify them. This approach is innovative because, in the work analyzed, we did not find a solution that uses the prediction of opcodes as input to a family classifier. The family classifier obtained an average accuracy of 91%.

1 INTRODUÇÃO

Mesmo com o 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* apareceram durante os 8 primeiros meses de 2018 (RAJ SAMANI, 2018).

Malware são *software* projetados com intenções maliciosas e podem ser usados para espionagem, extorsão, sabotagem ou para executar tarefas não autorizadas (RAYMOND J. CANZANESE, 2015). *Malware* podem ser dos mais variados tipos: *worms*, *trojans*, *rootkits*, *spyware*, *adware*, etc. (GAVRILUT et al., 2009).

O método de detecção por assinatura, que é um método clássico para detecção de vírus, utiliza um banco de dados com assinaturas digitais de *malware* para realizar a busca de partes do código malicioso em arquivos. Porém, técnicas que alteram levemente o código malicioso burlam os antivírus com relativa facilidade. A criação de métodos para detectar *malware* através de seu comportamento, por outro lado, é bastante complicada (UCCI et al., 2017).

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 (AHMADI et al., 2016).

O trabalho de um analista, devido a essa evolução de códigos maliciosos, tem ficado cada vez mais complexo, cansativo e custoso. É fundamental, então, a implementação de soluções e ferramentas que facilitem o trabalho do analista e soluções automatizadas para tarefas de análise, classificação e detecção do *malware* contribuem de forma efetiva.

Neste trabalho é proposta uma arquitetura baseada na predição de *opcodes* para fazer a classificação de um *malware* em famílias. Essa abordagem é inovadora no sentido de que, em nenhum trabalho analisado, encontrou esse tipo de solução.

1.1 MOTIVAÇÃO

Identificar uma ameaça ao sistema é de extrema importância para a segurança da informação, contudo isso pode não ser uma tarefa simples para um analista, uma vez que existe uma grande quantidade de *malware* na rede e um time de especialistas não dispõe

de tempo hábil para tratar diversos incidentes e analisar muitos artefatos. A abordagem utilizando aprendizado de máquina para processar grande quantidade de artefatos permite que o analista dedique seu tempo para análises mais profundas quando necessário.

Na literatura existem alguns trabalhos que buscam resolver esse problema com abordagens semelhantes. Em Pascanu et al. (2015), o autor utiliza redes neurais recorrentes na classificação do artefato, empregando diferentes bibliotecas de *Python* para aprendizado profundo além de utilizar análise dinâmica do comportamento do artefato. Ainda, em McLaughlin et al. (2017) o autor aborda a identificação de *malware* para *Android*, utilizando análise estática, por meio do uso dos *opcodes* do artefato. Essa abordagem por meio de *opcodes* será utilizada neste trabalho. Contudo, tal trabalho utiliza redes neurais convolucionais, diferentemente deste que usa redes neurais recorrentes. No IME também foram realizados trabalhos na área, como por exemplo em de Andrade (2013), o autor faz uma análise dinâmica, usando a aplicação de técnicas de *sandbox* alinhado com aprendizado de máquina na identificação de artefatos, obtendo acurácia de 93% em uma base com 4.717 *malware* e 5.383 de *goodware*. Ainda, em Mangialardo (2015), o autor trabalhou unindo análise estática e dinâmica com aprendizado de máquina, realizando a justaposição das características estáticas e dinâmicas escolhidas previamente, obtendo acurácia de 93% em uma base com 3.633 *malware* e 2.659 de *goodware*. Por último, em Pinto (2018) o autor realizou um trabalho aplicando aprendizado profundo, assim como neste projeto, para classificar *malware* utilizando aprendizado não supervisionado, abordando uma análise estática e trabalhando também com os *opcodes* de arquivos binários, obtendo Macro F1 de 93% com a mesma base de dados desse trabalho.

1.2 OBJETIVOS

O presente trabalho tem a finalidade de elaborar, implementar e testar um algoritmo que utiliza redes neurais para a tarefa de classificação de *malware*. Inicialmente, com uma base de dados escolhida, serão extraídas sequências de *opcodes* de diferentes famílias de *malware*. Essas sequências serão usadas para treinar, para cada família, um modelo que utiliza LSTM e tentará prever *opcodes* presentes nas amostras. Depois de treinados, os modelos serão usados para realizar previsões em diferentes amostras e, com os resultados das acurácias de quantos *opcodes* foram preditos corretamente, as amostras serão classificadas. Para classificar cada amostra, foram escolhidos duas formas de seleção: Um modelo com votação simples, onde a família do preditor que obtiver maior acurácia é escolhida para classificar a amostra, e um modelo utilizando redes neurais, que utiliza todos

os resultados dos preditores sobre uma amostra para procurar relações e classifica-la.

1.3 JUSTIFICATIVA

A Estratégia Nacional de Defesa, aprovada pelo decreto nº 6.703, de 18 de dezembro de 2008, especifica o setor cibernético como prioritária para o Exército. Dentro do Exército, o Instituto Militar de Engenharia está inserido nas áreas de Ciência e Tecnologia e de Recursos Humanos, tornando-se responsável por formar engenheiros qualificados e produzir pesquisas sobre tecnologia da informação e, mais especificamente, na área de segurança na informação.

Visando expandir os conhecimentos sobre análise de *malware* e *deep learning*, o tema foi proposto para ampliar os conhecimentos dentro e fora do Exército na criação de métodos automatizados na análise de *malware*, buscando independência tecnológica na área de segurança da informação.

Este trabalho surgiu como uma complementação do trabalho de doutorado do TC Sant’Ana que tem como objetivo definir uma arquitetura de aprendizado profundo aplicada à área de análise de *malware* que permita correlacionar o resultado de saída da rede e com as entradas por meio de técnicas de interpretabilidade.

Dentro do Exército e em outras instituições nacionais, a criação de métodos mais ágeis e eficientes proporciona uma busca por independência tecnológica no Brasil, que se está entre os 20 países que mais enfrentam riscos de segurança cibernética. (KASPERSKY, 2018). Tecnologias nacionais são mais seguras e confiáveis, uma vez que seu projeto é conhecido.

1.4 METODOLOGIA

O trabalho teve início com um estudo acerca de aprendizado de máquina, especificamente sobre aprendizado profundo e funcionamento de redes neurais recorrentes, juntamente com um estudo sobre análise de *malware*. Também foi realizada a preparação do ambiente de experimentos, bem como a instalação das ferramentas necessárias para a execução do trabalho.

A partir disso, foi desenvolvida uma prova de conceito utilizando rede recorrente (LSTM) empregando a biblioteca para *Python* voltada para aprendizado profundo *Keras*, que consolida os conhecimentos adquiridos na fase inicial, avaliando a possibilidade de dar continuidade ao trabalho de predição de *opcodes*. Além disso, foi feita a análise exploratória da base de dados, voltada para a extração de *opcodes* da base de dados

e a execução das atividades de pré-processamento, com o tratamento dos dados para alimentar os modelos e a codificação dos *opcodes*.

Foi realizada uma codificação de *opcodes* simples, numerando de 0 até a quantidade de *opcodes* distintos extraídos, em seguida foi realizada uma modelagem inicial da montagem da rede neural recorrente para cada família de *malware* e o treinamento das mesmas, mas os resultados dos modelos apresentaram um desempenho ruim, então os *opcodes* foram novamente codificados usando a técnica *Word2vec*. Com os modelos treinados e com um desempenho aceitável, foram executados testes de classificação em famílias utilizando primeiramente um classificador com um sistema de votação simples e depois uma rede neural de classificação, juntamente com a análise dos resultados e comparação com o estado da arte. A elaboração do relatório acompanhou todo o andamento do trabalho.

1.5 ESTRUTURA DO TEXTO

No capítulo 2 será apresentada a fundamentação teórica. O capítulo definirá *malware* com seus tipos e variações e descreverá o processo de análise. Além disso serão introduzidos o conceito de redes neurais e, mais especificamente, de LSTM. No capítulo 3, será apresentada a proposta para o trabalho. No capítulo 4, será mostrado como foi feita a extração dos *opcodes*. O capítulo 5 aborda a aplicação do algoritmo de predição e a forma como foram codificadas as instruções para serem lidas pelos modelos. No capítulo 6, dois métodos para a classificação são descritos e comparados mostrando resultados com outros trabalhos. Por fim, segue-se a conclusão.

2 REFERENCIAL TEÓRICO

2.1 MALWARE

Malware é um *software* criado para ter efeitos indesejados ou danosos a um computador e oferecer grande ameaça para a segurança de sistemas de computadores (YUXIN; SIYI, 2019). Na criação de um *malware*, podem ser utilizadas várias técnicas. Uma bem simples é inserir partes de código em um arquivo de programa para que ele seja executado em algum computador de forma diferente e infecte o alvo (*Trojan*). As mais complexas envolvem criar um algoritmo sofisticado para ofuscar um binário malicioso e escondê-lo de anti-vírus (SAEED et al., 2013).

Esses códigos maliciosos são possíveis devido a erros em programas, como problemas na lógica ou má administração da memória no programa (KUMAR et al., 2009). Esses erros ocorrem porque nem sempre os programas são testados levando em consideração diferentes tipos de tentativas de burlar a segurança do programa. O *off-by-one* é um erro bem comum em que o programador erra alguma conta por 1. Um exemplo clássico é o *openSSH*, que foi criado para ser um programa de comunicação por terminal feito para substituir serviços inseguros como *telnet*, *rcp* e *rsh*. No código, existia a linha do código 2.1:

CÓDIGO 2.1: Código do *openSSH* com erro

```
if ( id < 0 || id > channels_alloc ) {
```

O código correto deveria ser aquele apresentado no código 2.2:

CÓDIGO 2.2: Código do *openSSH* corrigido

```
if ( id < 0 || id >= channels_alloc ) {
```

O erro apresentado no código 2.1 cria uma vulnerabilidade que permite que um usuário comum ganhe controle administrativo do sistema (ERICKSON, 2008).

Um *Malware* se aproveita de erros como citado anteriormente, além da baixa segurança dos sistemas para causar danos a computadores e redes. Cada *malware* se comporta de diferentes formas e tira proveito de diferentes tipos de erros (vulnerabilidades) em servidores, computadores pessoais ou, até mesmo, de sistemas de controle de usinas, como é o caso do *stuxnet*, criado para atacar o sistema *SCADA* de usinas de enriquecimento de

urânio iranianas. O *malware*, o qual foi classificado como *worm*, foi capaz de atrasar a produção da usina de *Bushehr*.

2.1.1 TIPOS E VARIAÇÕES DE *MALWARE*

Os vários tipos de estruturas e comportamentos de códigos maliciosos permitem agrupá-los em várias categorias. De acordo com Sung et al. (2005), os *malware* podem ser classificados em três gerações baseando-se no *payload*, método de propagação e permissão de vulnerabilidades:

- a) A **Primeira Geração** é composta por *malware* que se assemelham a vírus e se propagam via e-mail, compartilhamento de arquivos ou ação humana, ou seja, executar, em computadores da empresa, arquivos desconhecidos que, na verdade, são códigos maliciosos capazes de se replicar e se espalhar pela rede;
- b) Na **Segunda Geração**, os *malware* compartilham propriedades de *worms*. Se propagam pela *internet* sem a necessidade de ação humana e automaticamente realizam a varredura de vulnerabilidades. Combinam o comportamento de vírus e *trojans*;
- c) *Malware* da **Terceira Geração** são criados especificamente para atacar regiões geográficas ou organizações específicas. Empregam múltiplos vetores de ataque e exploram vulnerabilidades conhecidas e desconhecidas para afetar produtos ou tecnologias.

Além disso, de acordo com o comportamento dos *malware* (SIKORSKI; HOGIN, 2012), pode-se classificá-los em:

- a) **Worm** ou **Vírus** são códigos maliciosos que se replicam e infectam computadores diversos;
- b) **Backdoors** se instalam em computadores e permitem um acesso remoto ao atacante. Permitem realizar uma conexão com pouca ou nenhuma autenticação e executar comandos no sistema local;
- c) **Botnet** são similares a *backdoors*, porém se espalham por vários alvos. Todos os computadores afetados recebem a mesma instrução de controle do servidor atacante;
- d) **Downloader** permite a inserção de outro código malicioso através de conexões de rede. São comumente instalados no primeiro acesso de um atacante;

- e) ***Information-stealing malware*** coleta informações sobre o computador da vítima e as envia para o atacante. Um exemplo é o *keylogger*, que registra tudo o que é digitado no computador e depois envia para o *hacker*;
- f) ***Laucher*** é utilizado para iniciar outro código malicioso. Comumente utilizam técnicas não tradicionais para inicializar um código de maneira furtiva para garantir um acesso melhor ao computador;
- g) ***Rootkit*** são códigos maliciosos utilizados para esconder outros códigos. São pareados com outros *malware* como *backdoor* para garantir acesso remoto a máquina da vítima e dificultar a detecção;
- h) ***Scareware*** são códigos que incitam a vítima a comprar algum produto falso. Possuem uma interface que se assemelha a de um antivírus ou outro programa semelhante informando que o computador está infectado;
- i) ***Spam-sending malware*** é um código malicioso que infecta o computador da vítima e o usa para enviar *spams*.

Devido a evolução das técnicas de detecção de *malware* dos antivírus, por meio de assinaturas digitais de executáveis, ou seja a identificação do *malware* por meio do resultado de uma função *hash*, ou da análise do comportamento do binário malicioso, os atacantes criam continuamente *malware* cada vez mais complexos capazes de burlar a detecção de antivírus por meio de técnicas de codificação.

Devido a evolução citada, foram criadas as seguintes variações de *malware*:

- a) ***Malware Polimórficos*** são códigos que aparentam ser diferentes cada vez que se replicam, porém mantém o código intacto. Essa técnica consiste em criptografar o código original e juntá-lo a um módulo que realiza a descriptografia durante sua execução. Apenas alterando a ordem das instruções, um novo método de criptografia/descriptografia pode ser criado. Como o corpo do vírus está codificado, não é possível realizar a análise e o código de mutação pode gerar inúmeras rotinas de codificação; (MATHUR; HIRANWAL, 2013)
- b) ***Malware Metamórficos*** geralmente usam técnicas de codificação diferentes pois eles são capazes de alterar seu corpo através de inserção de instruções desnecessárias como a inserção de instruções *NOP*, por transposição de código ou por substituição de instruções equivalentes; (CHRISTODORSCU; JHA, 2004)

- c) **Malware Oligomórficos** possuem vários decodificadores ao invés de um. Isso significa que eles são capazes de se modificar de uma variante para outra.

A capacidade de codificação e alteração do código é um dos fatores que gera a grande quantidade de novos *malware* a cada ano e eleva o nível de dificuldade para detecção e análise dos artefatos. A seção seguinte fala sobre técnicas de análise presentes no estado da arte porém, elas não são suficientes para suprir toda a demanda de segurança.

2.2 ANÁLISE 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 infectadas. Nessa tarefa, é importante determinar exatamente o que um artefato pode fazer, como detectá-lo e como medir e conter os danos causados. (SIKORSKI; HOGIN, 2012)

Além da análise, a detecção e a classificação de *malware* são duas tarefas executadas por companhias de antivírus.

2.2.1 FASES DA ANÁLISE DE MALWARE

Durante a análise do *malware*, os executáveis são analisados para verificar se exibem comportamento malicioso, isso pode ser verificado através da busca por assinaturas ou análise de comportamento. Se o resultado for positivo, será atribuída a família de *malware* mais apropriada de acordo com um mecanismo de classificação (AHMADI et al., 2016). Com todas as informações 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 Sikorski e Hogin (2012), pode-se subdividir as análises estática e dinâmica em básica e avançada.

As quatro fases, como mostra a figura 2.1, 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.

2.2.1.1 ANÁLISE ESTÁTICA BÁSICA

Na análise estática, o artefato é analisado sem ser executado procurando-se por padrões maliciosos em sua estrutura. Pode confirmar se um código é malicioso, obter informações

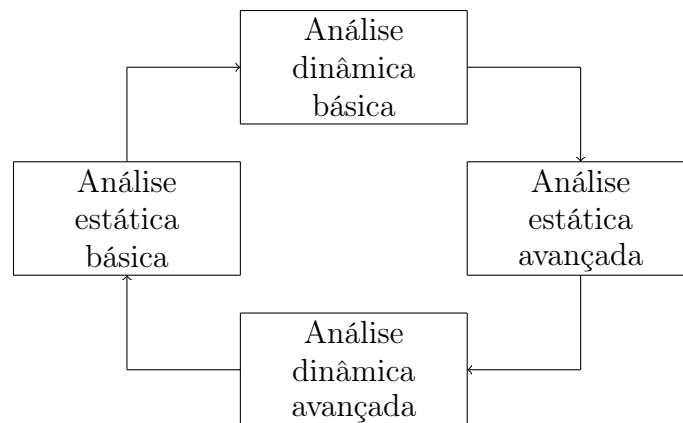


FIG. 2.1: Fases da análise de *malware*

básicas de sua funcionalidade e prover aspectos sobre possíveis conexões em redes. Por exemplo, por meio da aplicação *Dependency Walker*, o analista é capaz de listar funções chamadas dinamicamente por um executável e, dessa forma, é possível analisar quais *DLLs* são importadas e é possível prever certos comportamentos do artefato. A utilização da função *URLDownloadToFile* da *DLL Urlmon.dll* por um *malware* indica que o comportamento dele envolve o *download* de algum arquivo. É uma fase rápida e direta mas é ineficaz contra *malware* mais sofisticados, os quais são implementados utilizando técnicas de ofuscação, que serão comentadas adiante.

2.2.1.2 ANÁLISE DINÂMICA BÁSICA

Consiste em executar o *malware* em um ambiente virtual controlado como *sandbox* ou máquina virtual e observar seu comportamento. *Sandbox* é um *container* que emula um processo e mantém tudo o que foi gerado dentro desse *container*. Máquina virtual emula um sistema completo e mantém todas as informações dentro dele. Com ferramentas previamente instaladas, é possível verificar quais processos são executados, quais conexões de rede o vírus tenta criar e, no caso de o sistema afetado ser o *Windows*, é possível analisar quais *DLLs* são chamadas e quais chaves de registro são modificadas. A medida que o analista descobre novas informações sobre o *malware*, assim como nas outras fases, ele produz um relatório contendo tudo sobre o que foi descoberto, dando maior suporte para o início de outra fase da análise.

2.2.1.3 ANÁLISE ESTÁTICA AVANÇADA

Essa fase é feita realizando-se a engenharia reversa do código do *malware* com o auxílio de um *disassembler* e analisando as instruções em *assembly* para descobrir o que o programa

faz sem executá-lo.

Dessa forma, a análise das instruções é importante pois pode revelar comportamentos do artefato que não foram, por exemplo, executados durante a fase de análise dinâmica.

No entanto, essa técnica é mais complexa que as demais e exige do analista um conhecimento profundo em programação *assembly* e conceitos específicos sobre sistemas operacionais nos quais *malware* se encontra.

2.2.1.4 ANÁLISE DINÂMICA AVANÇADA

A análise dinâmica avançada é realizada com um *debugger* para examinar o estado interno de um executável e chamar cada instrução do código para descobrir o que exatamente o código faz. Esse tipo de análise é feita quando se deseja obter informações difíceis de serem obtidas a partir das outras técnicas. Além disso, essa fase pode ser considerada complexa, uma vez que o analista deve conhecer o *assembly* bem como a estrutura de pilha e registradores, juntando conhecimento da fase de análise estática avançada e de dinâmica básica.

2.2.2 TÉCNICAS DE DETECÇÃO DE *MALWARE*

Existem dois principais métodos para a detecção de *malware*:

- a) **Método baseado em Assinatura:** É a técnica mais rápida e eficaz para detectar *malware* conhecidos. (GRIFFIN et al., 2009). Na fase de geração, uma sequência grande o suficiente de *bytes* do artefato malicioso é selecionada de forma a representar unicamente o mesmo: assinatura do vírus. Na fase de detecção, o antivírus analisa um executável procurando por assinaturas. Se a assinatura for descoberta, o executável é tratado como malicioso. Embora eficaz, essa técnica é bastante fraca contra *malware* polimórficos ou metamórficos, pois para cada variante deve-se gerar uma assinatura.
- b) **Método baseado em Comportamento:** Esse método identifica as ações do executável ao invés de procurar por padrões no código, identificando programas que, apesar de possuírem códigos diferentes, possuem o mesmo comportamento. Uma assinatura comportamental pode identificar várias amostras de *malware* e esse tipo de detecção auxilia na identificação de códigos maliciosos capazes de se codificar.

As técnicas citadas são típicas de antivírus, porém são ineficazes quando consideramos o grande aumento na quantidade de *malware*, já que ambas as técnicas dependem de um

analista para gerar a assinatura ou a heurística.

2.2.3 TÉCNICAS ANTI-ANÁLISE E ANTI-DETECÇÃO

A ofuscação de *malware*, que modifica o artefato sem interferir no comportamento do executável, torna difícil a detecção por meio da detecção por assinaturas.

A criação de métodos para detectar *malware* através da semântica, por outro lado, é bastante complicado (UCCI et al., 2017).

Além disso, existem quatro técnicas anti-engenharia reversa descritas em (SIKORSKI; HOGIN, 2012).

2.2.3.1 ANTI-DISASSEMBLY

Anti-disassembly são técnicas feitas manualmente pelo autor do *malware* criando códigos especiais para que as ferramentas de *disassembly* gerem uma sequência incorreta de instruções. O método se aproveita do fato de que essas ferramentas realizam suposições (durante o processo de *disassembly*) e possuem limitações.

Anti-disassembly é utilizado para prolongar ou prevenir a análise de um código. Técnicas de engenharia reversa podem ser usadas, mas o nível de conhecimento necessário por parte do analista é maior.

Além de prevenir a análise humana, técnicas *anti-disassembly* são efetivas, principalmente, contra análises automatizadas. Vários algoritmos de detecção de *malware* utilizam *disassemblers* para identificar e classificar o *malware*, porém esses algoritmos não são capazes de burlar a técnica de *anti-disassembly*.

2.2.3.2 ANTI-DEBUGGING

Anti-debugging são técnicas utilizadas por *malware* para reconhecer que estão sobre o controle de *debuggers* e dificultar a análise. O *malware* que utiliza essa técnica é capaz de alterar o caminho normal do código ou modificar código para causar erro na execução e pará-lo.

A técnica mais simples usa funções da API do *windows* para checagem. A função **IsDebuggerPresent** que busca na estrutura do **Process Enviroment Block** (PEB) pelo campo **IsDebugged** e retorna 1 em caso positivo.

Outras técnicas envolvem procurar manualmente na estrutura de memória por informações do *debugger*, identificar o comportamento do *debugger* procurando por *breakpoints*

ou execução de instruções passo-a-passo e até interferir no funcionamento do *debugger* com interrupções e exceções colocados no código.

2.2.3.3 ANTI-VIRTUAL MACHINE

Da mesma forma que as anteriores, as técnicas de *anti-Virtual Machines* tentam detectar o uso de máquinas virtuais e, em caso positivo, comportam-se no sentido de dificultar a análise. Por exemplo, eles podem se comportar de forma diferente ou simplesmente não executar sua funcionalidade principal. Técnicas de *anti-Virtual Machine* são encontradas em *malware* comuns como *bots*, *scarewares* e *spywares*.

Essas técnicas se baseiam em procurar, no sistema operacional, por indícios de que o sistema está instalado em um ambiente virtualizado como, por exemplo, procurar pelo nome da placa de rede típica do autor da solução de virtualização.

A popularidade de códigos maliciosos capazes de realizar essas ações tem diminuído: provavelmente por causa do aumento do uso de virtualização em ambientes de produção. Ou seja, não é por que o *malware* está em uma máquina virtual que ele está sendo analisado.

2.2.3.4 OFUSCAÇÃO

A ofuscação¹ de *malware* é realizada por programas chamados *packers*², que modificam o artefato sem interferir no comportamento do executável, tornam difícil a detecção por meio de antivírus e a análise do artefato de forma manual pelo analista. A análise estática do código de interesse não é possível em um artefato empacotado. Além disso, a detecção e classificação desses *malware* utilizando assinatura digital é muito mais complicada, pois é qualquer assinatura de código viável seria do código do *packer*.

Packers utilizam um algoritmo de compressão para transformar o executável em um outro executável que armazena o primeiro executável como dados e adiciona um desempacotador que é iniciado pelo sistema operacional quando o *malware* é executado. O arquivo de saída é comprimido, encriptado ou transformado de forma a tornar sua análise mais difícil.

¹Em Sikorski e Hogin (2012), a quarta técnica descrita é o uso de empacotadores para criptografar o conteúdo do *malware*. A ofuscação tem um sentido mais amplo, pois todas as outras técnicas anti-análise descritas nas seções anteriores podem também ser consideradas ofuscação, uma vez que dificultam a leitura da sequência de instruções que o *malware* executa. Neste trabalho, consideraremos ofuscação como o uso de empacotadores

²empacotadores ou codificadores

Além disso, os empacotadores ainda podem também implementar outras técnicas anti-análise como *anti-VM*, *anti-debugging* e *anti-disassembly* no processo e podem codificar todo o código do executável ou apenas parte dele.

Com a grande variedade e o grande crescimento do número de *malware*, as técnicas mais utilizadas para análise e detecção de programas maliciosos tem se tornado cada vez mais ultrapassadas. A detecção por meio de assinatura digital exige banco de dados cada vez mais complexos e maiores. A análise de grande quantidade de informação exige um esforço enorme das companhias. Aliado a isso, existe o aumento do número de famílias de *malware* e de novas variantes dentro da família, o que dificulta o trabalho de analistas (AHMADI et al., 2016). Além disso, a criação de métodos para detectar *malware* através da semântica, por outro lado, é bastante complicada (UCCI et al., 2017). O método de análise proposto nesse trabalho se baseia na análise estática e, portanto, não é adequado para *malware* que implementa ofuscação. No entanto ele é interessante para *malware* que implementa *anti-VM*, pois o artefato não precisa ser executado.

2.3 APRENDIZADO PROFUNDO

Durante muito tempo almejou-se que o computador fosse capaz de resolver problemas complexos assim como o cérebro humano. Dessa forma, surgiu o aprendizado profundo. Problemas complexos nesse caso são situações que fogem do escopo onde a programação é eficiente, velocidade em cálculos aritméticos e execução de instruções. Para resolver esse tipo de problemas é necessário um aprendizado baseado em exemplos, o que é conhecido como aprendizado de máquina. No aprendizado de máquina é criado um modelo para aplicar o que foi aprendido com os exemplos e seus resultados podem variar de acordo com a escolha de algoritmos de classificação (BUDUMA; LOCASCIO, 2017). O aprendizado profundo é derivado do aprendizado de máquina, o qual introduz o *perceptron* multi-camada, que mapeia um conjunto de entradas em uma saída, e cada camada tem o intuito de tratar um nível de informação da entrada (GOODFELLOW et al., 2016), o que torna o aprendizado profundo é exatamente o *perceptron* multi-camada, também pode ser interpretado como a profundidade da camada intermediária da rede neural (LÖFWANDER, 2017). Além disso, o aprendizado profundo trás também o que é chamado de engenharia de características automatizadas, o qual não necessita de um analista gerenciando quais características terão relevância para o modelo. O próprio modelo interpreta quais recursos devem ser utilizados (KOEHRSEN, 2018), que são processados em cada camada da rede neural, incorporando padrões com o fluxo de informação pela rede (LÖFWANDER,

2017), como é ilustrado na figura 2.2.

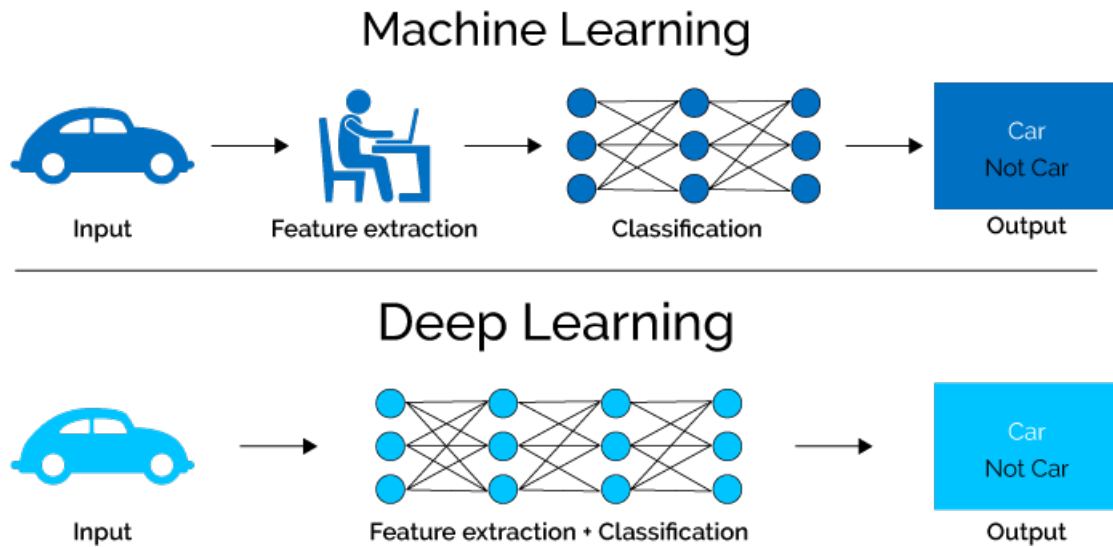


FIG. 2.2: Diferença entre aprendizado de máquina e aprendizado profundo (ODI; NGUYEN, 2018)

2.3.1 REDES NEURAIS

Como foi dito, o aprendizado profundo foi pensado para imitar o pensamento humano, separando o fluxo de informação em várias camadas, para tal redes neurais foram criadas para simular o funcionamento de vários neurônios. Assim foi formulado um modelo para representar um neurônio, que recebe várias entradas e cada uma possui um tipo diferente de relevância, terminando com outras saídas. Formalizando a esquematização do

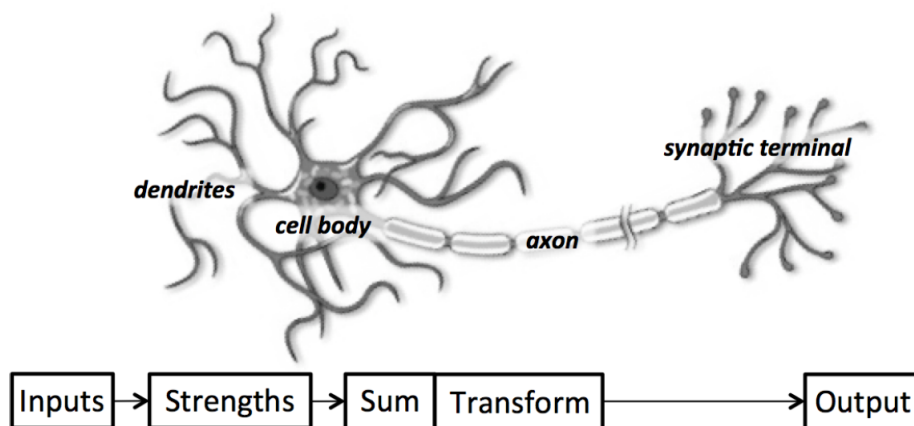


FIG. 2.3: Descrição funcional da estrutura do neurônio (BUDUMA; LOCASCIO, 2017)

neurônio temos um vetor i com as entradas, um vetor w com os pesos de cada entrada, representando a relevância de cada entrada, e obtemos $o = f(i \cdot w + b)$, onde o são as

saídas e b é chamado de bias e regula a influência dos pesos na saída. Combinando vários neurônios artificiais temos uma rede neural. A rede neural é composta por camada de

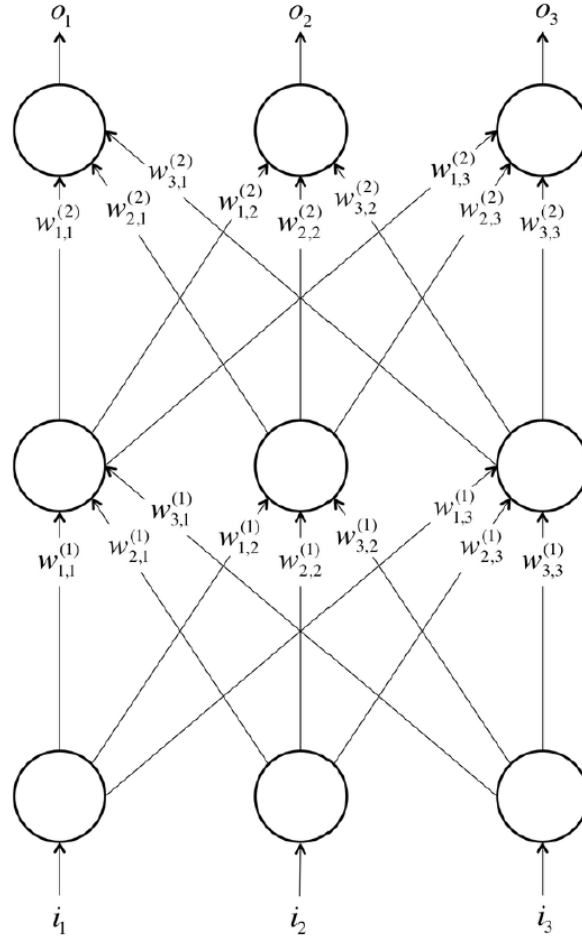


FIG. 2.4: Exemplo de rede neural com três camadas com três neurônios por camada(BUDUMA; LOCASCIO, 2017)

entrada, camadas intermediárias e camada de saída (BUDUMA; LOCASCIO, 2017).

2.3.2 BACKPROPAGATION

Em uma rede neural a informação da entrada i se propaga pelas unidades intermediárias para gerar um resultado. Esse formato é conhecido como propagação progressiva. Contudo calcular o gradiente da função custo, afim de minimizá-lo, não é simples e por isso foi proposto o método de *backpropagation*. O método consiste em balancear os pesos da rede por meio do resultado de cada amostra do treino a partir da comparação com a saída esperada, tendo assim os pesos da penúltima camada balanceados. Esse balanceamento é feito novamente na camada anterior e vai se propagando até a camada de entrada. Esses pesos são referentes a um caso de treino. O gradiente mínimo produz a média dos pesos

encontrados em todos os casos de treino (GOODFELLOW et al., 2016).

2.3.3 REDES NEURAIS RECORRENTES

Redes neurais recorrentes são um tipo específico de redes neurais, os quais 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, implicando que quanto mais a informação flui pela camada intermediária, maior a quantidade de composições de funções de ativação. Ao calcular o gradiente descendente, devido a regra da cadeia, quanto maior a sequência de composição de funções de ativação menor é o gradiente. Isso resulta em ao adicionar novas entradas com o tempo, menor é o impacto das entradas passadas na atualização dos pesos, esse fenômeno é chamado de *vanishing gradient*. Para solucionar esse problema foi desenvolvida a arquitetura *long short-term memory* (LSTM), que tem o objetivo de manter as informações importantes aprendidas pela rede neural com o passar do tempo, feito através da célula de memória.

2.3.4 LONG SHORT-TERM MEMORY - LSTM

Na arquitetura da uma LSTM existem 3 portas: entrada, esquecimento e 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 (JONES, 2017). Comparando a arquitetura LSTM com uma rede neural recorrente regular, podemos ver,

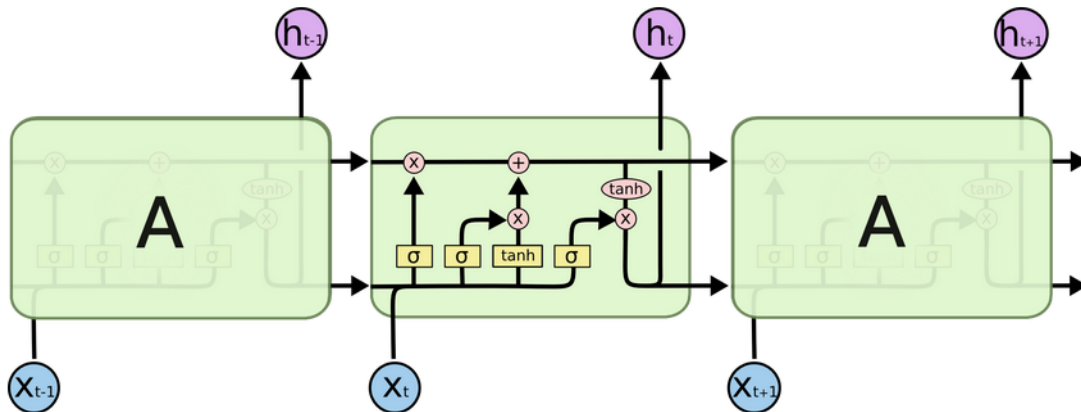


FIG. 2.5: Camada LSTM expandida (OLAH, 2015)

devido a forma linear que a informação se propaga com o tempo, que a LSTM não tem

o seu gradiente atenuado com um número grande de estados (BUDUMA; LOCASCIO, 2017).

2.3.4.1 ASPECTOS MATEMÁTICOS

A lógica dentro das portas citadas podem ser equacionadas conforme as equações em 2.1,

$$\begin{aligned}
 f_t &= \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \\
 i_t &= \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \\
 \tilde{C}_t &= \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \\
 C_t &= f_t * C_{t-1} + i_t * \tilde{C}_t \\
 o_t &= \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \\
 h_t &= o_t * \tanh C_t
 \end{aligned} \tag{2.1}$$

onde $*$ é o produto elemento por elemento e a função σ é definida pela equação 2.2:

$$\sigma(x) = \frac{1}{1 + \exp(-\lambda x)} \tag{2.2}$$

onde λ é o parâmetro de inclinação da função sigmóide. Além disso os termos são localizados na célula da figura 2.6:

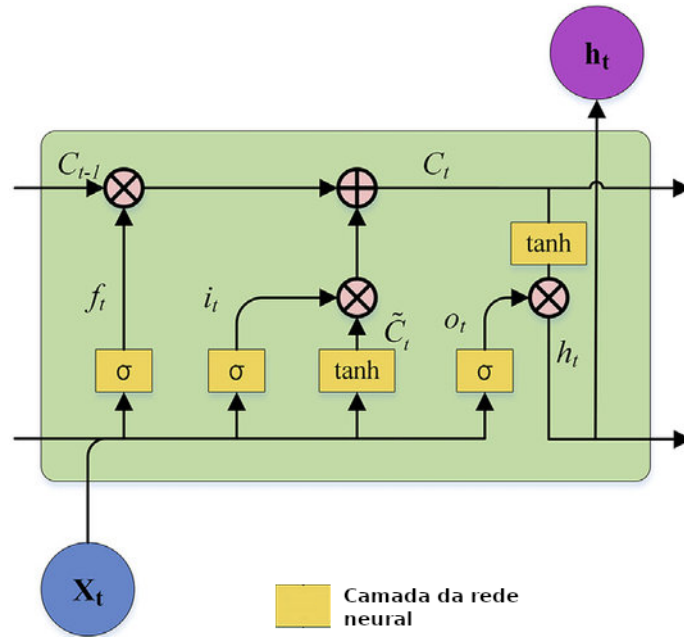


FIG. 2.6: Célula LSTM detalhada (LI et al., 2017)

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 citada anteriormente.

Já i_t indica o que deve ser atualizado, agindo como a camada da porta de entrada, e \tilde{C}_t elenca novas informações candidatas a serem adicionadas. Combinando i_t e \tilde{C}_t temos a atualização do estado C_{t-1} para C_t .

O termo o_t define o que a saída que irá influenciar nos próximos estados, caracterizando a porta de saída. Os termos b_f , b_i e b_o 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 ligados ao aprendizado, sendo esses parâmetros balanceados com o intuito de diminuir o erro e obter um modelo eficiente (OLAH, 2015).

2.3.4.2 APLICAÇÃO

Ao aplicar os conceitos em um conjunto de dados, durante a definição da arquitetura da rede é necessário definir alguns hiper-parâmetros próprios da implementação da LSTM na biblioteca *Keras*.

- a) *lookback*: indica quantas etapas anteriores devem ser importante na predição da próxima etapa (KOMPELLA, 2018).
- b) *batch size*: indica o número de exemplos de treinos usados em uma iteração do treino, uma vez que a rede neural não pode tratar todos os exemplos de treino de uma vez.
- c) *epoch*: representa uma varredura pelo conjunto inteiro de dados, treinando o modelo. Faz-se necessário usar mais de um *epoch*, pois o treinamento na parte final do conjunto pode aumentar muito erro na parte inicial (SHARMA, 2017).
- d) *time step*: reflete o ponto de observação da amostra, ou seja qual o tamanho da sequência de amostras vai alimentar o modelo
- e) otimizadores: tem função acelerar o processo de treino produzindo resultados compatíveis. O otimizador padrão mais usado é o *adam*, existem outros que são utilizados em casos específicos de conjunto de dados (BROWNLEE, 2017).

3 UMA PROPOSTA DE PREDIÇÃO DE OPCODES PARA A CLASSIFICAÇÃO DE MALWARE EM FAMÍLIAS

Este projeto tem a proposta de, a partir da base de dados disponibilizada pela *Microsoft*³ de 2015, classificar os *malware* em famílias por meio da predição de *opcodes*, ou instruções em *assembly*, e com isso ser capaz de classificá-lo quanto sua taxonomia. A base de dados possui nove famílias.

A primeira parte do trabalho se dividiu em quatro fases: (1) extração e tratamento de dados a partir de uma base de dados escolhida contendo diversas classes de *malware*; (2) treinamento de preditores utilizando LSTM, gerando um modelo de predição de *opcodes* para cada família; (3) validação do modelo LSTM de predição de *opcodes* no conjunto de teste; e (4) avaliação do treinamento, onde é verificado se o resultado produzido pelo modelo proposto para predição de *opcodes* é adequado para alimentar um classificador de famílias (próxima parte do trabalho). A figura 3.1 apresenta um resumo desta primeira parte do trabalho.

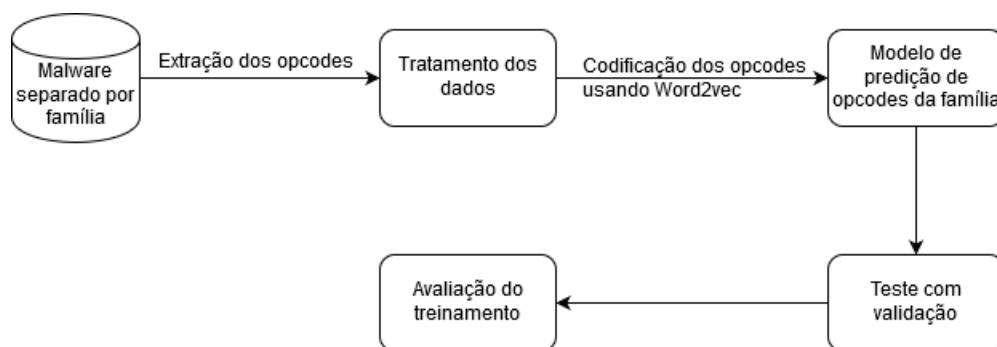


FIG. 3.1: Diagrama da primeira parte do trabalho

Na segunda parte do trabalho, a partir dos resultados das acurácias produzidas pelo modelo de predição de *opcodes*, foram desenvolvidas duas soluções para a classificação em família: um sistema de votação e uma rede neural artificial. Para uma dada amostra de *malware*, o sistema de votação apenas escolhe o modelo que produziu a melhor acurácia na predição e, portanto, não necessita de treinamento. A rede neural artificial foi treinada a partir do resultado de acurácias produzidas pelos modelos de predição para uma parte do conjunto de treinamento. A partir de então, para uma dada amostra de *malware*, a rede neural utiliza as acurácias dos 9 modelos preditores de *opcodes* para realizar a classificação

³<https://www.kaggle.com/c/malware-classification/data>

em família. A figura 3.2 apresenta um resumo da segunda parte do trabalho. O vetor de acurácias está representado com 9 valores de Acc 1 a Acc 9.

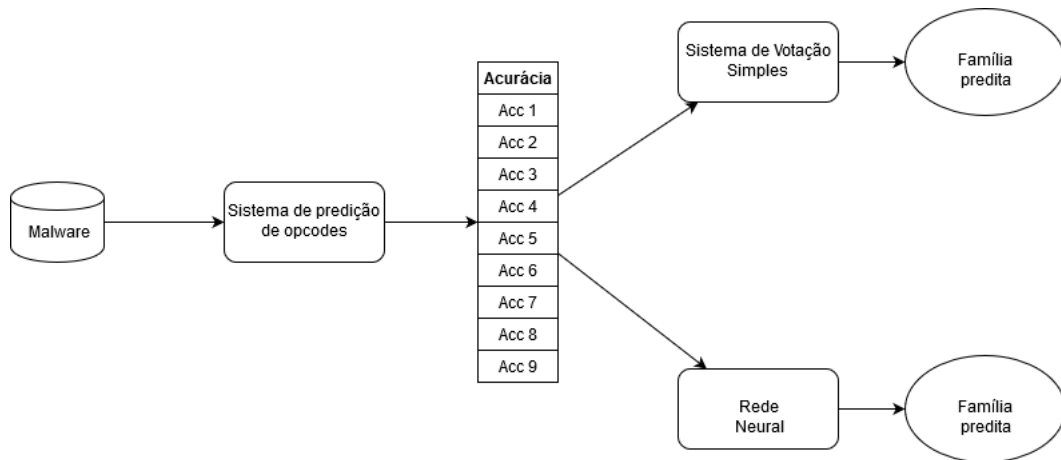


FIG. 3.2: Diagrama da segunda parte do trabalho

Para finalizar, a figura 3.3 mostra uma visão geral do processo de classificação de um *malware*. Inicialmente, os *opcodes* são extraídos e processados (codificados). Em seguida, a sequência codificada é analisada pelos nove modelos de predição de *opcodes*, um para cada família. Cada modelo apresenta como saída a acurácia de predição de *opcodes* para a amostra de *malware*. Finalmente, os resultados de acurácias são transmitidos para um classificador, que pode ser um sistema de votação simples ou uma rede neural, que produz o resultado da classificação.

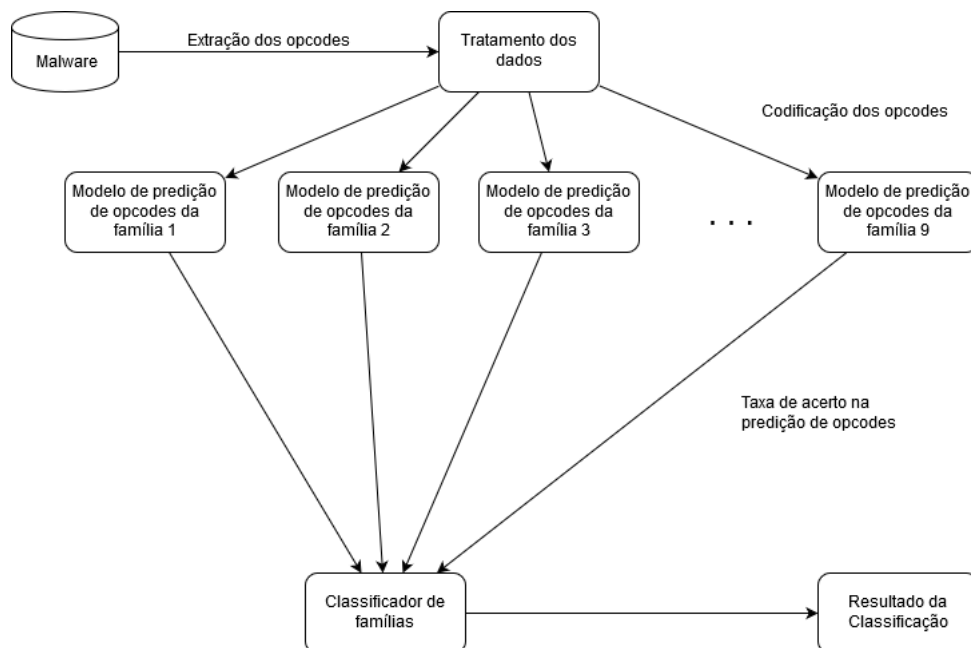


FIG. 3.3: Diagrama da fase de aplicação dos modelos

4 EXTRAÇÃO DE OPCODES

A base de dados utilizada foi obtida nos arquivos do Desafio de Classificação de *Malware* da *Microsoft*⁴ de 2015 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. A tabela 4.1 mostra a quantidade de amostras por classe.

Classe	Nome da Classe	Número 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

TAB. 4.1: Número de amostras em cada classe

A extração das sequências de *opcodes* foi realizada para que essas sequências servissem de entrada para o treinamento dos modelos de preditores das famílias.

4.1 EXTRAÇÃO DE INSTRUÇÕES

Os arquivos do tipo *asm* disponibilizados pela *Microsoft* foram gerados pelo *software IDA-Pro*. Dessa forma, os binários originais de cada *malware* não eram conhecidos, pois tiveram seus cabeçalhos removidos o que inviabilizou o uso de bibliotecas como *pefile* ou *capstone*, da linguagem *python* que são capazes de extrair as instruções de executáveis. Assim, para extrair os *opcodes* foi desenvolvido um *script* capaz de processar os arquivos *asm* do tipo texto e exportados pelo IDA, e extrair os *opcodes*.

Arquivos executáveis para *Windows* são do tipo *pefile* e possuem diversas seções, cada uma com uma finalidade, para que seja possível sua execução. Normalmente, a seção *.text* possui o código em *assembly* que será executado e a seção *.data*, geralmente, contem os dados, como valores inteiros, *strings* entre outras variáveis. A nomenclatura

⁴<https://www.kaggle.com/c/malware-classification/data>

dessas seções é uma convenção e portanto não é obrigatória. A listagem 4.1 tirada do arquivo *21l6VM9tKEuCUhL7DjeY.asm*⁵ do tipo texto simples, mostra uma sequência de instruções presentes no arquivo *asm*.

No código exportado pelo IDA Pro, as instruções em *assembly*, como *push*, *add* e *mov* estão localizadas na mesma coluna do arquivo texto. O *script* em *python* criado utilizou-se desse fato para realizar a extração dos *opcodes*.

CÓDIGO 4.1: Exemplo de instruções assembly presentes nos malwares

```
.text:00401000 55                push  ebp
.text:00401001 8B EC            mov   ebp, esp
.text:00401003 83 EC 5C         sub   esp, 5Ch
.text:00401006 83 7D 0C 0F      cmp   [ebp+Msg], 0Fh
.text:0040100A 74 2B            jz    short loc_401037
.text:0040100C 83 7D 0C 46      cmp   [ebp+Msg], 46h
.text:00401010 8B 45 14         mov   eax, [ebp+iParam]
.text:00401013 75 0D            jnz   short loc_401022
.text:00401015 83 48 18 10      or    dword ptr [eax+18h], 10h
.text:00401019 8B 0D A8 3E 42 00 mov   ecx, dword_423EA8
.text:0040101F 89 48 04         mov   [eax+4], ecx
```

Dessa forma, o *script* em *python* precisava apenas procurar pela seção *.text:* e ler a instrução, que se inicia no caractere 80 de cada linha.

4.2 PROBLEMAS NA EXTRAÇÃO

A extração não é uma tarefa fácil, uma vez que o código executável nem sempre se encontra na seção *.text*. Na tabela 4.2 são apresentadas as quantidades de *malware* separadas por classe e por seção onde se encontra o código executável.

⁵No código 4.1, os espaços entre o nome da seção, *.text:* e as instruções foram alterados para que coubessem na página. Originalmente, as instruções estão na coluna 80 de cada linha do arquivo *asm*

Classe	.text	CODE	seg000	seg001	.zenc	UPX0
1	1513	19	0	1	0	0
2	2278	0	2	0	191	0
3	2936	0	5	0	0	0
4	447	0	22	0	0	0
5	21	5	0	0	0	0
6	294	438	0	16	0	0
7	387	0	11	0	0	0
8	1169	9	12	1	0	28
9	1011	1	0	0	0	0

TAB. 4.2: Número de amostras em cada classe com código em diferentes seções

A tabela 4.2 não exaure todas as nomenclaturas presentes na base de dados para seções com códigos. Além dos nomes presentes na tabela, foram encontradas amostras cujo código está presente em seções nomeadas como *.itext*, *.idata*, *.code*, *acggagg*, *yogmamm*. Dessa forma, o *script* teve que ser adaptado para incluir essas seções na extração.

Um possível motivo para essa variação de nomes da seção executável é que o autor do artefato quis dificultar a análise. Conforme tais seções executáveis foram identificadas, um novo *script* era criado para processá-las.

CÓDIGO 4.2: Função para procurar instruções em uma linha

```
def try_get_ins(line):
    global ins4
    global ins3
    global dd
    words = line.split("_")
    for word in words:
        if word in ins4 or word in ins3:
            for i in range(len(line)):
                try:
                    if line[i:i+4] in ins4:
                        return i
                    if line[i:i+3] in ins3:
                        return i
                except:
                    print("")
            if word in dd:
                return -1
    return 0
```

O código 4.2 mostra a função que procura, em cada linha, por uma instrução. Se ela for encontrada, retorna um valor positivo referente à posição da instrução na linha. Esse código será aplicado para procurar se alguma instrução existe na linha: em caso positivo, o *script* armazenará o nome da seção e, em seguida, lê novamente o arquivo extraindo os *opcodes* presentes na seção com código.

CÓDIGO 4.3: Laço para ler as linhas do arquivo

```
for source_line in in_:
    fixed = fix_line(source_line)
    maybe = fixed.split(":")[0]
    if ponto == 0:
        ponto = try_get_ponto(fixed)
    ins = try_get_ins(fixed)
    if ins > 0:
        text = maybe
        break;
    elif ins == -1:
        where_dd = maybe
        ins = 0
```

O laço no código 4.3 chama a função *try_get_ins*, apresentada no código 4.2, para cada linha presente na amostra depois de gravar o nome da seção com código na variável *maybe*. Quando a função retorna um valor positivo, o nome da seção é armazenado na variável *text*, que será usada posteriormente no *script* para procurar pelos *opcodes*. Assim, o *script* não precisa saber o nome da seção previamente e, portanto, pode extrair os *opcodes* de todos os *malware*, mesmo que a seção com código não tenha o nome *.text*.

A partir do conteúdo da variável *text*, o *script* extrai todas as instruções. Uma vez que a posição já está armazenada em *ins*, basta que a linha seja lida na posição até o próximo espaço presente na linha e, assim, todas as instruções são salvas.

O *script* foi capaz de extrair sequências de instruções de 10816 amostras. As 52 amostras restantes foram analisadas manualmente. Dessas amostras, 44 possuíam apenas instruções *db*, *dd* e *dw*, que não são necessárias para o preditor de código, uma vez que são apenas armazenamento de informação. A tabela 4.3 mostra quantas amostras de cada classe possuem apenas instruções *db*, *dd* e *dw*.

Classe	Numero de amostras
1	0
2	3
3	4
4	22
5	0
6	0
7	6
8	9
9	0

TAB. 4.3: Número de amostras em cada classe apenas com instruções db, dd, dw

As oito amostras restantes possuem apenas duas seções: *HEADER* e *GAP*. Nenhuma dessas seções possui nenhum tipo de instrução *assembly* e, portanto, essas amostras foram desconsideradas no estudo. Todas fazem parte da classe 1 e acredita-se que houve um problema no processo de *disassembly* realizado pela ferramenta *IDA Pro*. Pode-se inferir que, por alguma técnica *anti-disassembly* foi utilizada pelo criador do vírus e, assim, não foi possível retirar as instruções dessas amostras.

5 APLICAÇÃO DE ALGORITMOS DE PREDIÇÃO

5.1 ATIVIDADES DE PRÉ-PROCESSAMENTO

A primeira atividade de pré-processamento foi a extração de *opcodes*, os quais foram organizados em um arquivo *csv* que contém em cada linha: nome do arquivo, a família do *malware* e os *opcodes* extraídos do arquivo. Esse arquivo foi separado em 9 arquivos diferentes, uma para cada uma das 9 famílias citadas. Cada um dos arquivos contém, em cada linha, uma amostra. Na primeira célula da linha está o nome da amostra e, nas células seguintes, as instruções *assembly* presentes no código da amostra. Um exemplo de linha nesse formato referente a família 1 é: *01kcPWA9K2BOxQeS5Rju, push, push, mov* e assim em diante.

Cada arquivo foi separado em dois arquivos, um para o treino, contendo 80% das amostras de cada família, e outro para teste com o restante das amostras. Assim, das 10.816 amostras, 8.652 são do conjunto de treinamento e 2.164 são do conjunto de teste. O arquivo de treino foi utilizado para treinar o modelo LSTM de predição de *opcodes* de cada família e o modelo classificador de família que utiliza rede neural.

O número total de *opcodes* distintos presentes na base de dados é de 549. Inicialmente, para cada um desses *opcodes*, foi atribuído um número de 0 até 548 de acordo com a ordem alfabética. Contudo, essa codificação aproxima *opcodes* com sentidos contrários como *pop*, codificado como 320, e *push*, codificado como 357. Os modelos iniciais foram treinados com essa codificação e os resultados não foram satisfatórios, indicando que utilizar um vetor com apenas uma dimensão não é suficiente. Assim, uma nova forma de codificação foi necessária.

5.2 REPRESENTAÇÃO DOS *OPCODES*

Para resolver o problema de codificação citado na seção anterior, buscou-se uma alternativa utilizada para representar/codificar palavras na área de processamento natural de linguagem: o *Word2vec*. O *Word2vec* é um método para criar *embeddings* entre palavras, que são transformadas em vetores. Essa técnica é utilizada em *NLP* (*Natural Language Processing*, ou processamento de linguagens naturais), para prever palavras de textos

em linguagens naturais. Diversas ferramentas famosas utilizam o método: *Siri*, *Google Assistant*, *Alexa*.

Antes disso, os *NLP* utilizavam números reais para representar as palavras ou números inteiros, índices das palavras no vocabulário. Isso não permite criar similaridade entre as palavras e limita a performance de várias tarefas. (MIKOLOV et al., 2013)

O *Word2Vec*, como o nome sugere, tenta descrever palavras como forma de vetores. Em linguagem natural, muitos trabalhos envolvem vetores de 300 dimensões.

A ideia por trás desses vetores é que eles tentam agrupar grupos semânticos. Cada dimensão é entendida como uma característica da palavra. Na figura 5.1, são mostradas três palavras codificadas com word2vec aplicado ao idioma inglês. Os valores de cada dimensão de seus vetores foram substituídos por cores. Percebe-se que as palavras "*Man*" e "*Woman*" são mais parecidas e que "*King*" está mais distante de ambas. Além disso, a palavra "*King*" é muito mais parecida com "*Man*" do que com a palavra "*Woman*".

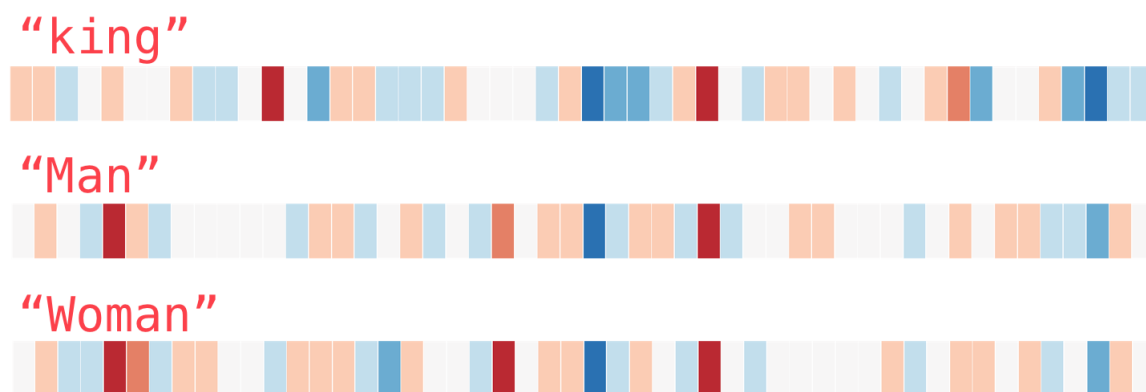


FIG. 5.1: Vetores de exemplos com cores (ALLAMAR, 2019)

Esse modelo não apenas representa graus de similaridade entre palavras, como também é possível prever, aproximadamente qual seria o vetor relacionado a uma certa palavra no vocabulário. Por exemplo, o vetor resultado da operação $\text{vetor}(\text{"King"}) - \text{vetor}(\text{"Man"}) + \text{vetor}(\text{"Woman"})$ é muito próximo do vetor("*Queen*"), essa aritmética pode ser interpretada como *King* está para *Man* assim como *Queen* está para *Woman*.

Embora tenham 300 dimensões, é possível visualizar os vetores em um gráfico com apenas duas dimensões, utilizando técnicas de redução de dimensionalidade, como por exemplo a análise de componentes principais, ou *Principal Component Analysis* - PCA. Assim, reduzindo para 2 dimensões é possível visualizar os vetores de forma mais intuitiva.

Na figura 5.2, é possível notar as duas relações importantes já citadas aplicadas a países e capitais.

Primeiramente, todos os pontos que representam países estão do lado esquerdo do gráfico e todas as capitais estão do lado direito do gráfico, mostrando como o processo de geração dos vetores separa os grupos semânticos presentes no vocabulário. Além disso, pode-se perceber que os vetores que levam de pontos que representam países para pontos que representam capitais é parecido, tanto em direção quando em comprimento.

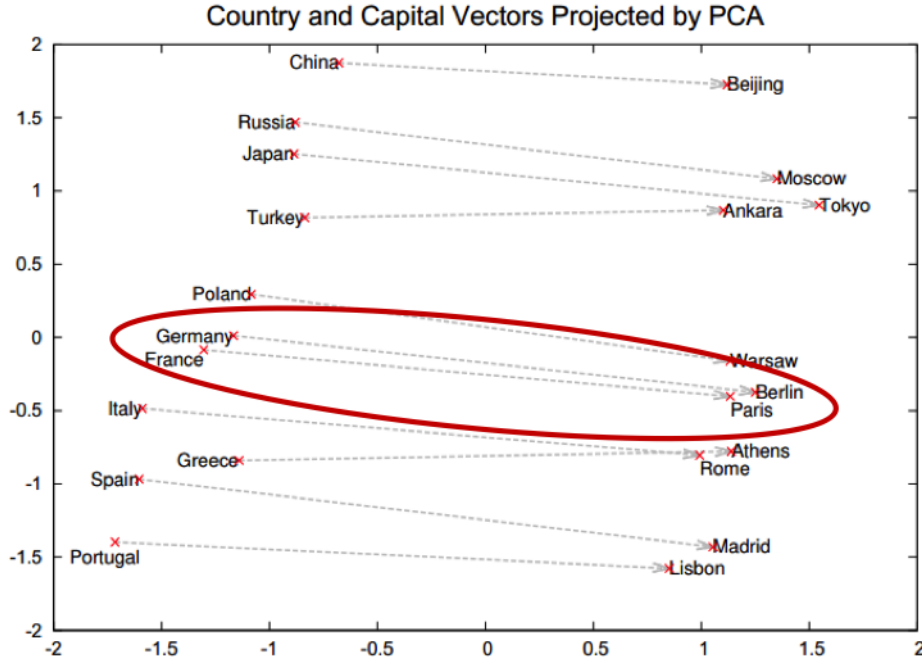


FIG. 5.2: Vetores de exemplos com países e capitais (GOM, 2015)

Neste trabalho foi utilizado uma codificação fruto da aplicação do método *Word2vec* às instruções *assembly*. Esses vetores foram gerados por Massarelli et al. (2019), criando uma base de dados de mneumônicos e operandos na forma de vetores. Mnemônicos são palavras reservadas de *assembly* que identificam cada comando e os operandos são os argumentos desse comando.

5.3 MODELAGEM UTILIZADA PARA OS PREDITORES

A partir dos vetores criados por Massarelli et al. (2019), 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 Massarelli et al. (2019).

Os *opcodes* extraídos do conjunto de treinamento e testes foram codificados utilizando

esse novo vetor. Se algum dos 549 *opcodes* extraídos do conjunto de treinamento ou teste não for encontrado, a codificação utilizada será do vetor *UNKNOWN*(MASSARELLI et al., 2019).

Para observarmos como essa codificação representa a semântica do *opcode* apresentamos a figura 5.3 que mostra quatro instruções projetadas em 2D: *add* no ponto vermelho, *sub* no ponto verde, *pop* no ponto amarelo e *push* no ponto azul. Os eixos X e Y da figura 5.3 não tem significado pois são eixos de projeção do vetor de dimensão 100.

É possível ver que o vetor que vai do ponto vermelho até o ponto verde é semelhante ao vetor que vai do ponto azul para o ponto amarelo. Assim, os modelos veem as instruções *pop* e *push* como opostas da mesma forma que as instruções *add* e *sub*.

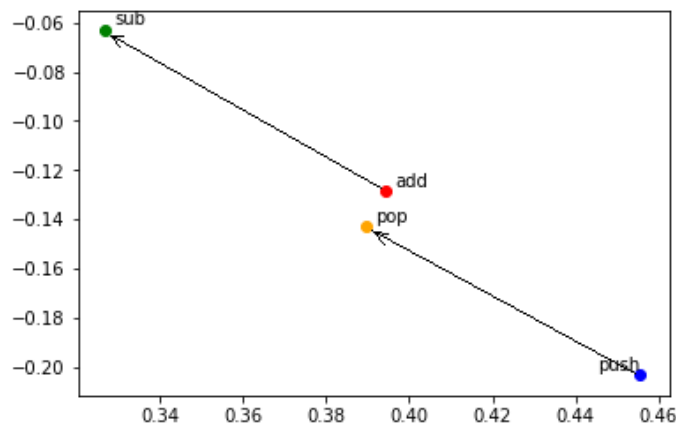


FIG. 5.3: Exemplo de vetores com dimensão reduzida de *opcodes* utilizados no trabalho

Com os *opcodes* codificados, foi criada, para o conjunto de treinamento e teste, uma matriz que agrupa os dados de forma que sequências de 5 *opcodes* consecutivos sejam a entrada e o sexto elemento seja a saída desejada. A quantidade de *opcodes* da sequência é conhecida como *look-back* e para este trabalho foi definido o valor de 5.

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) gera a linha da matriz (op1, op2, op3, op4, op5) que será usada para predizer op6.

Para adequar os dados ao formato de entrada da biblioteca *Keras*, a dimensão da entrada deve ser a seguinte: número de sequências de *opcodes*, número de *time steps*, dimensão da codificação do *opcode*. O número de sequências reflete a quantidade de amostras que serão processadas por vez (*batch size*). O *time step* é 5, conforme explicado anteriormente e, finalmente, a dimensão da codificação dos *opcodes* é 100 pois foi utilizada a codificação *Instruction2vec* gerada por Massarelli et al. (2019).

CÓDIGO 5.1: Montagem e treinamento do modelo

```
time_step = 5
look_back = 5
epoch = 100
units = 100
batch = 100
model = Sequential()
model.add(LSTM(units, input_shape=(time_step, 100)))
model.compile(loss='mean_squared_error', optimizer='adam')
for k in range(epoch):
    treinoF = open("treino" + label + ".txt", "r")
    for linha in treinoF:
        listalinha = (linha.split("\n")[0]).split(",")
        malwareopcodes = []
        for i in range(len(listalinha)):
            elemento = listalinha[i]
            malwareopcodes.append(
                vetor[opcodes == opcodes[int(float(elemento))]] [0])
        malwareopcodes = numpy.asarray(malwareopcodes)
        a, b = create_dataset(malwareopcodes, look_back)
        history = model.fit(a, b, epochs=1, batch_size=batch)
    treinoF.close()
```

5.4 DEFINIÇÃO E TREINAMENTO DOS MODELOS

O modelo preditivo de cada família foi criado com 100 unidades LSTM, definindo a dimensão dos pesos, os quais determinam o aprendizado dos portões da célula LSTM já abordados. Valores maiores criam maior quantidade de pesos consequentemente aumentam a complexidade do modelo. Além disso, o número de unidades define a dimensão de saída do modelo, no caso de previsão de *opcodes* é necessário uma saída compatível com a codificação adotada.

A dimensão dos *opcodes* codificados e o *time step* compõe a dimensão da entrada do modelo. Além disso, o treinamento foi feito minimizando o erro quadrático médio, até a convergência do erro, e com *batch size* = 100, separando em lotes de tamanho 100 o conjunto de treino, conforme o código 5.1.

Foram executadas 100 épocas no treinamento com a condição de parada de cair em um mínimo local, já o *batch size* foi escolhido devido restrições de memória.

5.5 RESULTADOS DOS TREINAMENTOS

Conforme citado anteriormente, os resultados da predição de *opcodes* utilizando números de 0 a 548 para cada um dos 549 *opcoes* únicos identificados não obteve bom resultado. A

tabela 5.1 resume os resultados de previsão de *opcodes* para cada uma das famílias. A taxa de acerto de *opcodes* apresentada representa a execução das amostras de treinamento no modelo LSTM treinado. É visível que as taxas são muito baixas, com exceção da família 5, e concluiu-se que a codificação de *opcode* utilizada não era a adequada.

	Taxa de acerto de opcodes
Modelo 1	0.0040%
Modelo 2	1.1900%
Modelo 3	0.0060%
Modelo 4	0.0060%
Modelo 5	89.100%
Modelo 6	0.0001%
Modelo 7	0.0005%
Modelo 8	0.0300%
Modelo 9	0.0040%

TAB. 5.1: Avaliação do treino dos modelos usando codificação escalar

A partir de então, todos os modelos agora consideram a codificação de *opcodes* por *Opcode2vec* conforme já descrito anteriormente.

Os modelos de cada família obtidos do estudo preliminar apresentaram as curvas de aprendizado próximas à curva do modelo 4 na figura 5.4, com exceção do modelo 2 que devido a demora do seu treinamento teve um critério de parada mais rigoroso, com apenas 35 épocas como mostra a figura 5.5.

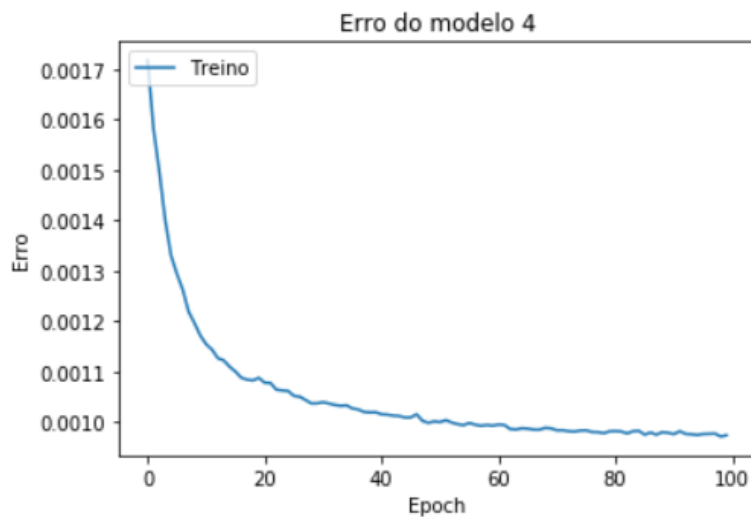


FIG. 5.4: Gráfico do erro do modelo 4 por quantidade de épocas

Em função da condição de parada ajustada no modelo 2, não foi observada a convergência do seu erro, como nos demais modelos. A acurácia de predição de *opcodes* foi

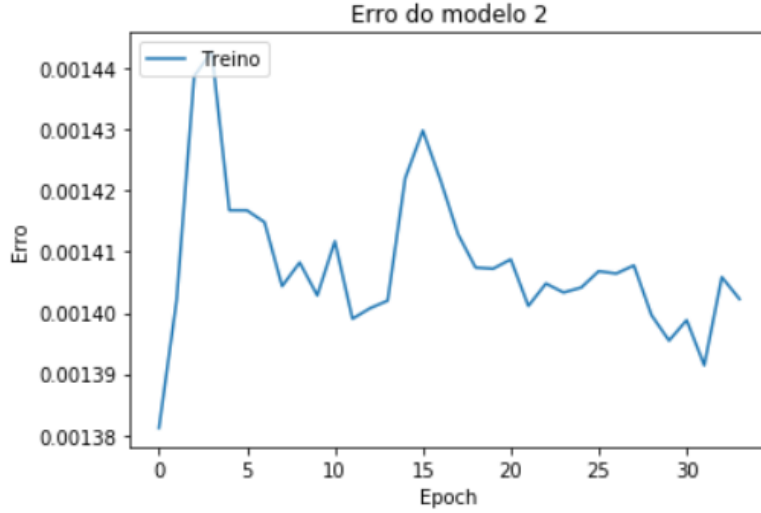


FIG. 5.5: Gráfico do erro do modelo 2 por quantidade de épocas

calculada para cada modelo. A taxa de acertos da predição de *opcodes* para o conjunto de testes é apresentado na tabela 5.2 dentro das respectivas classes.

O vetor de saída dos modelos é comparado com cada *opcode* codificado. A taxa de acerto de *opcodes* nos testes de cada modelo é apresentada na tabela 5.2.

	Taxa de acerto de <i>opcodes</i>
Modelo 1	40.6%
Modelo 2	37.9%
Modelo 3	73.4%
Modelo 4	53.6%
Modelo 5	94.0%
Modelo 6	91.3%
Modelo 7	69.1%
Modelo 8	80.3%
Modelo 9	90.1%

TAB. 5.2: Avaliação do treino dos modelos usando a codificação proveniente do Op-code2vec

Os piores desempenhos dos modelos 1 e 2 podem ter ocorrido devido a ambas as famílias possuírem arquivos com maiores quantidades de *opcodes* e que boa parte desses *opcodes* não representa funcionalidade específica da família. Como esperado, a acurácia na predição de *opcodes* em todos os modelos utilizando codificação *Word2vec* ficou bastante superior quando comparadas com a predição de *opcodes* utilizando uma representação escalar de *opcodes*.

6 DEFINIÇÃO E TREINAMENTO DO MODELO DO CLASSIFICADOR DE FAMÍLIAS

Após o treino dos modelos utilizando LSTM, cada um dos nove modelos realizou predições sobre cada uma das nove famílias e as acurácias dos preditores foram salvas em um total de 81 arquivos *csv* (9 modelos de famílias x 9 acurácias por modelo), que continha apenas uma linha com os valores das acurácias.

Com os resultados das predições, dois tipos de soluções foram testadas para classificar os *malware*: a primeira utiliza um sistema de votação simples e a segunda utiliza uma rede neural treinada com os resultados dos preditores por amostra.

Para testar os classificadores, foram utilizadas 60% do conjunto de amostras selecionado para os testes nos preditores, totalizando um total de 1.289 amostras.

6.1 SISTEMA DE VOTAÇÃO SIMPLES

Nessa abordagem, cada uma das amostras de teste (1.289 amostras) foi submetida com os nove diferentes modelos. O maior valor entre as acurácias era utilizado para classificar as amostras na sua respectiva família.

A tabela 6.1 mostra a matriz de confusão desse modelo.

		Família Predita								
		Fam 1	Fam 2	Fam 3	Fam 4	Fam 5	Fam 6	Fam 7	Fam 8	Fam 9
Família Verdadeira	Fam 1	169	0	1	0	3	1	0	10	0
	Fam 2	234	2	0	0	3	30	0	18	9
	Fam 3	0	0	352	0	0	0	0	0	0
	Fam 4	1	0	0	52	1	0	0	0	0
	Fam 5	1	0	0	0	3	0	0	0	0
	Fam 6	1	0	0	1	3	35	0	20	28
	Fam 7	9	0	32	0	0	0	0	5	0
	Fam 8	17	0	0	0	3	0	0	124	0
	Fam 9	4	0	45	0	3	0	0	0	69

TAB. 6.1: Matriz de confusão do sistema classificador de votação simples para 1.289 amostras de teste.

Em cada linha são mostradas as amostras de cada família. As colunas representam como o modelo classificou as amostras. Dessa forma, o valor presente na posição da linha n e coluna m diz quantas amostras da família n foram classificadas como família

m. O primeiro valor da tabela, 169, presente na linha 1 e coluna 1, diz a quantidade de amostras da família 1 que foram classificadas corretamente. Os valores em negrito são as classificações corretas do modelo.

Ainda na tabela 6.1, é possível perceber que o preditor de *opcodes* da família 1 tem altas taxas de acerto em todas as famílias, uma vez que apenas na família 3 possui zero amostras as quais o preditor não possui acurácia maior que os demais preditores. Na família 2, mais de 80% das amostras foram classificadas como família 1, indicando que essa forma de classificação não é adequada.

É possível obter os resultados gerais do sistema utilizando a função *classification_report* da biblioteca *keras*. O relatório gerado para o sistema de votação está na tabela 6.2.

Famílias	<i>precisão</i>	<i>abrangência</i>	<i>f1-score</i>	<i>quantidade</i>
Família 1	0.39	0.92	0.55	184
Família 2	1.00	0.01	0.01	296
Família 3	0.82	1.00	0.90	352
Família 4	0.98	0.96	0.97	54
Família 5	0.16	0.75	0.26	4
Família 6	0.53	0.40	0.45	88
Família 7	0.00	0.00	0.00	46
Família 8	0.70	0.86	0.77	144
Família 9	0.65	0.57	0.61	121
<i>acurácia</i>			0.63	1289
<i>média</i>	0.58	0.61	0.50	1289
<i>média ponderada</i>	0.73	0.63	0.54	1289

TAB. 6.2: Resultados gerais do sistema de votação

Na tabela 6.2, a precisão de cada família é a relação entre o número de amostras classificadas corretamente e o número total de amostras que o classificador julgou ser dessa família. Por exemplo, na família 4, 52 amostras foram classificadas corretamente e uma amostra da família 6 foi classificada incorretamente como família 4. Nesse caso, a precisão é de $52/(52 + 1) = 0.98$.

O valor *abrangência* de cada família é a relação entre o número de amostras classificadas corretamente e o número de amostras da mesma família classificadas incorretamente. Por exemplo, das 4 amostras da família 5, 3 foram classificadas corretamente e 1 amostra foi classificada incorretamente. Assim, o valor de *abrangência* é $3/(3 + 1) = 0.75$.

De acordo com Narkhede (2018), a fórmula usada para calcular o valor de *f1-score* é:

$$F_1 - score = 2 * \frac{precisão * abrangência}{precisão + abrangência}$$

$F1$ -score é uma medida de desempenho comumente utilizada para classificações com múltiplos rótulos (FUJINO et al., 2008). É calculada com a média harmônica dos valores *abrangência* e precisão, ou seja, ela busca um equilíbrio entre esses valores (ROCCA, 2019), penalizando-o pelo menor dos valores.

É importante notar, ainda na tabela 6.2, que a acurácia do modelo foi de 63%.

Devido aos resultados insatisfatórios do sistema de votação, uma nova abordagem mais complexa foi necessária. Um modelo que utiliza rede neural pode ir além da escolha do modelo que teve a melhor acurácia na predição de *opcode* e classificar melhor as amostras.

6.2 CLASSIFICADOR COM REDE NEURAL

Para a classificação com uma rede neural, foi utilizada uma rede *feed-forward* densa com três camadas, ou seja, cada neurônio de uma camada se conecta com todos os neurônios da camada seguinte. A figura 6.1 mostra a estrutura da rede neural utilizada. A primeira coluna, com 9 nós, representa as informações de entrada para o modelo, que serão melhor explicadas mais adiante nesta seção. A segunda coluna de nós é a camada de entrada na rede contendo 12 neurônios. Em seguida, a camada oculta possui 10 neurônios e a camada de saída possui 9 neurônios.

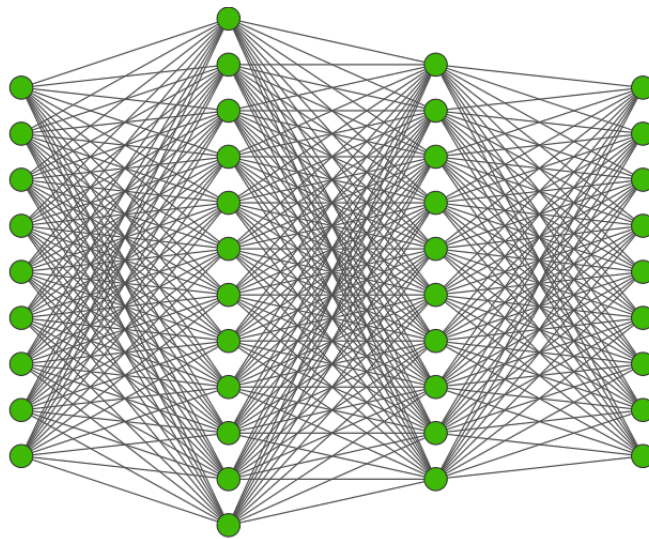


FIG. 6.1: Estrutura da rede neural utilizada no classificador

Na última camada é utilizada uma função de ativação chamada de *softmax*, que força as saídas de cada neurônio a representar a probabilidade dos dados serem de classes definidas. Assim, a soma de todas as nova saídas da rede será 1.

Por questão de limite de tempo de execução e dos prazos do trabalho, para o treinamento da rede foram utilizadas 20% do conjunto de treinamento original (80% de todas as amostras) totalizando 1.718 amostras. Nesse contexto, as amostras de entrada representam as 9 saídas dos 9 modelos de preditores de *opcodes* onde cada posição possui o valor da acurácia de um deles.

A predição do classificador foi comparada com o valor da família verdadeira codificado com o *one-hot encoder*. Dessa forma, a rede aprendia as relações entre as acurácias de forma a classificar corretamente os *malware*. Para o teste do classificador, foi utilizado o mesmo conjunto do sistema de votação com 1.289 amostras.

Foram realizados experimentos com os seguintes valores de hiper-parâmetros:

- *batch_size*: 2,5,10 e 20;
- *epochs*: 500,1000 e 1500;
- Função de perda: *binary cross entropy* e *mean squared error*.

De todos os experimentos, a rede que obteve melhores resultados possuía os seguintes hiper-parâmetros:

- *epochs* = 1500;
- *batch_size* = 2;
- Função de perda: *mean squared error*.

Os resultados da classificação da rede utilizando os hiper-parâmetros acima estão descritos na matriz de confusão apresentada na tabela 6.3.

A matriz de confusão apresentada para o classificador baseado em rede neural apresenta melhoras significativas quando comparado ao sistema de votação simples. Houve uma pequena queda na taxa de acertos nas famílias 1, 4, 5 e 8. Nessas famílias, o número de classificações corretas diminuiu em 3, 1, 1 e 1 amostras, respectivamente. Porém, nas outras famílias, a taxa de acertos aumentou significativamente. As famílias 2 e 7 apresentaram taxas muito mais altas que no sistema de votação, comprovando que a rede neural é capaz de relacionar os resultados das acurácias para prever a família verdadeira do *malware*.

O relatório gerado pela função *classification_report* está na tabela 6.4. O classificador acertou 91% da classificação de todas as amostras. A família 5 obteve o pior resultado com apenas 50% das amostras classificadas corretamente. Isso se deve ao fato de que

		Família Preditá								
		Fam 1	Fam 2	Fam 3	Fam 4	Fam 5	Fam 6	Fam 7	Fam 8	Fam 9
Família Verdadeira	Fam 1	166	8	1	1	0	1	1	6	0
	Fam 2	17	275	0	0	0	1	0	2	1
	Fam 3	0	0	352	0	0	0	0	0	0
	Fam 4	0	1	0	51	0	1	1	0	0
	Fam 5	0	1	0	0	2	0	0	0	1
	Fam 6	5	2	0	2	0	77	0	1	1
	Fam 7	0	5	0	0	0	0	41	0	0
	Fam 8	11	8	0	0	0	0	2	123	0
	Fam 9	4	29	0	0	0	3	0	0	85

TAB. 6.3: Matriz de confusão do sistema classificador baseado em rede neural com hiper-parâmetros *epochs*=1500, *batch size*=2 e função de perda MSE para 1.289 amostras de teste.

existiam poucas amostras em relação as outras famílias. Dessa forma, a rede não foi capaz de aprender completamente as relações das acurácias na família.

Famílias	<i>precisão</i>	<i>abrangência</i>	<i>f1-score</i>	<i>quantidade</i>
Família 1	0.82	0.90	0.86	184
Família 2	0.84	0.93	0.88	296
Família 3	1.00	1.00	1.00	352
Família 4	0.94	0.94	0.94	54
Família 5	1.00	0.50	0.67	4
Família 6	0.93	0.88	0.90	88
Família 7	0.91	0.89	0.90	46
Família 8	0.93	0.85	0.89	144
Família 9	0.97	0.70	0.81	121
<i>acurácia</i>			0.91	1289
<i>média</i>	0.93	0.84	0.87	1289
<i>média ponderada</i>	0.91	0.91	0.91	1289

TAB. 6.4: Resultados gerais do classificador utilizando rede neural

Comparando as tabelas 6.2 e 6.4, é possível ver que a maioria dos os valores de *precisão*, *abrangência* e *f1-score* aumentaram. De acordo com (ROCCA, 2019), valores altos de *precisão* e *abrangência* significam que a classe está bem definida no modelo.

Na tabela é possível verificar que, com exceção dos valores *abrangência* das famílias 9 e 5, todos os valores estão em torno de 90%. E, como resultado, a acurácia do modelo completo é de 91%, ou seja, 50% melhor que o sistema de votação.

A título de comparação, resolveu-se realizar uma análise dos resultados obtidos neste trabalho e compará-los com os resultados obtidos em Pinto (2018) e Kaggle (2015).

As tabelas 6.5 e 6.6 são resultados obtidos por Pinto (2018) e pelo modelo vencedor

do desafio realizado pela *Microsoft*, ambas retiradas de Pinto (2018).

Em Pinto (2018), o autor realiza a mesma extração de *opcodes* da base de dados da *Microsoft*. Em seguida uma codificação dos *opcodes* foi feita utilizando o *Term Frequency - Inverse Document Frequency* - TF-IDF (MANNING et al., 2008). O TF-IDF tenta indicar a importância de uma palavra (*opcode*) de um binário em relação a uma coleção de binários.

A classificação é feita por meio de uma rede autocodificadora pré-treinada para reconstruir as entradas TF-IDF. A saída da parte codificadora dessa rede autocodificadora é usada como entrada para uma rede neural para classificação em famílias. Desta forma, pode-se observar que essa solução possui as mesmas limitações de classificação de um artefato em uma família ao utilizar como entrada apenas as informações de *opcodes* da seção executável do *PEfile*. Ao mesmo tempo, os resultados passam a ser comparáveis com o que foi desenvolvido neste trabalho.

A tabela 6.5 apresenta os resultados obtidos pelo classificador gerado por Pinto (2018). Podemos observar que os valores de F1-score são, 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.

Famílias	<i>precisão</i>	<i>abrangência</i>	<i>f1-score</i>	<i>quantidade</i>
Família 1	0.9786	0.9482	0.9632	386
Família 2	0.9713	0.9823	0.9767	620
Família 3	0.9986	0.9918	0.9952	736
Família 4	0.9640	0.8992	0.9304	119
Família 5	0.7273	0.7273	0.7273	11
Família 6	0.8916	0.9628	0.9258	188
Família 7	0.9794	0.9500	0.9645	100
Família 8	0.9216	0.9577	0.9393	307
Família 9	0.9718	0.9488	0.9602	254
<i>acurácia</i>			0.9986	2721
<i>média</i>	0.9337	0.9297	0.9314	2721
<i>média ponderada</i>	0.9676	0.9669	0.9670	2721

TAB. 6.5: Métricas por classe para o classificador final – Rede 4, 4 camadas ocultas, conjunto de dados de Bigramas. (PINTO, 2018)

Finalmente, a tabela 6.6 apresenta o resultado obtido pelo vencedor da disputa do *Kaggle*. É interessante observar que todos os resultados de F1-score são bem superiores aos apresentados neste trabalho. Deve-se aqui, no entanto, destacar que o vencedor do *Kaggle*

utilizou-se de características como contagem de *opcodes*, (2,3 e 4 gramas), intensidade de pixel, 4-grama byte, frequência de bytes simples, nome de funções, características derivadas extraídas de todo arquivo e não somente da seção executável. É natural que essa abordagem tenha potencial para extrair uma quantidade maior de informações quando comparadas a extração apenas de *opcodes*.

Famílias	<i>precisão</i>	<i>abrangência</i>	<i>f1-score</i>	<i>quantidade</i>
Família 1	0.9974	1.00	0.9987	1541
Família 2	0.9996	0.9992	0.9994	2478
Família 3	0.9997	1.00	0.9998	2942
Família 4	1.00	1.00	1.00	475
Família 5	1.00	0.9286	0.963	42
Família 6	0.9973	0.9987	0.998	751
Família 7	1.00	1.00	1.00	398
Família 8	0.9959	0.9976	0.9967	1228
Família 9	0.998	0.9941	0.996	1913
<i>acurácia</i>			0.9981	10868
<i>média</i>	0.9986	0.9909	0.9946	10868
<i>média ponderada</i>	0.9985	0.9982	0.9983	10868

TAB. 6.6: Métricas por família para a solução ganhadora do concurso *Microsoft/Kaggle* (KAGGLE, 2015)

7 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 arquivos tipo texto exportados pelo IDA, ao invés de usar bibliotecas especializadas para essa extração. Tão importante quanto a correta extração de *opcodes* foi a correta codificação dos *opcodes*. Neste trabalho, resultados interessantes foram obtidos utilizando uma técnica de codificação de *opcodes* baseadas no *word2vec* utilizando uma base de codificação desenvolvida em Massarelli et al. (2019).

Foram montados 9 modelos preditores de *opcodes* usando rede neural recorrente baseada em LSTM: a capacidade de predição de *opcodes* dos 9 modelos foi adequada para desenvolver um classificador de família baseado na saída dos preditores de *opcodes*.

Duas foram as propostas de classificador: um sistema de votação simples e um classificador baseado em rede neural para aproveitar possíveis correlações não lineares entre as taxas de predições de *opcodes* corretas (acurácias) entre os modelos.

Os resultados apresentados pelo classificador foram satisfatórios, a matriz de confusão apresentada revela que o classificador identifica muito bem todas as famílias com exceção da família 5.

O melhor resultado obtido foi utilizando, na saída dos preditores de *opcodes*, uma rede neural para classificar as amostras em famílias, obtendo o resultado médio de 91%.

Como trabalhos futuros, pode-se sugerir:

- Treinar a rede de predição de *opcodes* com todas as amostras de treinamento e utilizar essas acurácias para treinar a rede neural. No trabalho foram usadas apenas 20% das amostras totais de treinamento.
- Extrair *opcodes* e operandos, pois dessa forma são obtidas mais informações para a predição dos *opcodes*. Para isso, seria necessário gerar uma base mais completa usando a técnica *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* (DI et al., 2018) ou *BERT* (DEVLIN et al., 2018);
- Ajustar alguns hiper-parâmetros como: o *timestep* para predição de *opcodes*, maior número de épocas de forma a garantir a convergência do erro e habilitar parâmetros como *early stop*. Neste trabalho, devido a limitações de tempo de processamento, não foi possível variar os hiper-parâmetros de forma desejada;
- Utilizar uma estrutura LSTM mais complexa, como LSTM empilhadas, sempre levando em conta a possibilidade de *overfitting* e maior duração dos treinamentos.

8 REFERÊNCIAS BIBLIOGRÁFICAS

- 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.
- ALLAMAR, J. The Illustrated Word2Vec. Disponível em: <<https://jalammar.github.io/illustrated-word2vec/>>. Acesso em: 27 março de 2019.
- BROWNLIE, J. Gentle Introduction to the Adam Optimization Algorithm for Deep Learning. Disponível em: <<https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/>>. Acesso em: 10 mai. de 2019.
- 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.
- 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.
- DE ANDRADE, C. A. B. Análise Automática de Malwares Utilizando as Técnicas de Sandbox e Aprendizado de Máquina. Disponível em: <<http://www.comp.ime.eb.br/pos/arquivos/publicacoes/dissertacoes/2013/2013-Cesar.pdf>>. Acesso em: 10 mai. de 2019.
- 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.
- 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.
- ERICKSON, J. **Hacking: the art of exploitation**. [S.l.]: No starch press, 2008.

- 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. **Anais...** [S.l.: s.n.], 2008. Disponível em: <<https://www.aclweb.org/anthology/I08-2116.pdf>>. Acesso em: 3 out. de 2019.
- 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.
- GOM, J. Mining English and Korean text with Python. Disponível em: <<https://www.stechstar.com/user/zbxe/AlgorithmPython/52575>>. Acesso em: 27 março de 2015.
- GOODFELLOW, I.; BENGIO, Y. ; COURVILLE, A. **Deep Learning**. [S.l.]: MIT Press, 2016.
- GRIFFIN, K.; SCHMEIDER, S.; HU, X. ; CHIUEH, T.-C. Automatic generation of strings signatures for malware detection. **12th International Symposium on Recent Advances in Intrusion Detection**, v. 5758, p. 101–120, 2009.
- JONES, M. T. Arquiteturas de aprendizado profundo. Disponível em: <<https://www.ibm.com/developerworks/br/library/cc-machine-learning-deep-learning-architectures/cc-machine-learning-deep-learning-architectures-pdf.pdf>>. Acesso em: 10 mai. de 2019.
- KAGGLE, T. Microsoft Malware Winners' Interview: 1st place, "NO to overfitting!". Disponível em: <<http://blog.kaggle.com/2015/05/26/microsoft-malware-winners-interview-1st-place-no-to-overfitting/>>. Acesso em: 29 setembro de 2019.
- KASPERSKY. Kaspersky Securty Bulletin 2018. Disponível em: <<https://securelist.com/kaspersky-security-bulletin-2018-statistics/89145/>>. Acesso em: 8 de maio de 2019.
- KOEHRSEN, W. Why Automated Feature Engineering Will Change the Way You Do Machine Learning. Disponível em: <<https://towardsdatascience.com/why-automated-feature-engineering-will-change-the-way-you-do-machine-learning-5c15bf188b96>>. Acesso em: 10 mai. de 2019.

- 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.
- 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.
- LI, X.; PENG, L.; YAO, X.; CUI, S.; HU, Y.; YOU, C. ; CHI, T. Long short-term memory neural network for air pollutant concentration predictions: Method development and evaluation. **Environmental pollution (Barking, Essex : 1987)**, v. 231, p. 997–1004, 2017.
- LÖFWANDER, S. About Artificial Intelligence, Neural Networks & Deep Learning. Disponível em: <<https://www.ayima.com/blog/artificial-intelligence-neural-networks-deep-learning.html>>. Acesso em: 2 out. de 2019.
- MANGIALARDO, R. J. Integrando as Análises Estática e Dinâmica na Identificação de Malwares Utilizando Aprendizado de Máquina. Disponível em: <www.comp.ime.eb.br/pos/arquivos/publicacoes/dissertacoes/2015/2015-Reinaldo.pdf>. Acesso em: 10 mai. de 2019.
- MANNING, C. D.; SCHÜTZE, H. ; RAGHAVAN, P. **Introduction to information retrieval**. New York, NY, USA: Cambridge University Press, 2008.
- 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.
- 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.
- MCLAUGHLIN, N.; MARTINEZ DEL RINCON, J.; KANG, B.; YERIMA, S.; MILLER, P.; SEZER, S.; SAFAEI, Y.; TRICKEL, E.; ZHAO, Z.; DOUPÉ, A. ; JOON AHN, G. Deep android malware detection. In: PROCEEDINGS OF THE SEVENTH ACM ON CONFERENCE ON DATA AND APPLICATION

- SECURITY AND PRIVACY, 7., CODASPY '17, doi:10.1145/3029806.3029823., 2017. **Anais...** New York, NY, USA: ACM, 2017, p. 301–308. Disponível em: <<http://doi.acm.org/10.1145/3029806.3029823>>. Acesso em: 10 mai. de 2019.
- MIKOLOV, T.; CHEN, K.; CORRADO, G. ; DEAN, J. Efficient estimation of word representations in vector space. **arXiv preprint arXiv:1301.3781**, v. 3, 2013. Disponível em: <<https://arxiv.org/pdf/1301.3781.pdf>>. Acesso em: 29 set. 2019.
- NARKHEDE, S. Understanding Confusion Matrix. Disponível em: <<https://www.ayima.com/blog/artificial-intelligence-neural-networks-deep-learning.html>>. Acesso em: 2 out. de 2019.
- ODI, U.; NGUYEN, T. Geological facies prediction using computed tomography in a machine learning and deep learning environment. In: PROCEEDINGS OF THE 2018 UNCONVENTIONAL RESOURCES TECHNOLOGY CONFERENCE, 6., 2018. **Anais...** [S.l.: s.n.], 2018. Disponível em: <<https://www.researchgate.net/publication/326834670>>. Acesso em: 2 out. de 2019.
- OLAH, C. Understanding LSTM Networks. Disponível em: <<https://colah.github.io/posts/2015-08-Understanding-LSTMs/>>. Acesso em: 10 mai. de 2019.
- PASCANU, R.; STOKES, J. W.; SANOSSIAN, H.; MARINESCU, M. ; THOMAS, A. Malware classification with recurrent networks. In: 2015 IEEE INTERNATIONAL CONFERENCE ON ACOUSTICS, SPEECH AND SIGNAL PROCESSING (ICASSP), 40., 2015. **Anais...** [S.l.: s.n.], 2015, p. 1916–1920.
- PINTO, D. R. Aprendizado Profundo Aplicado à Análise Estática de Malwares. Disponível em: <www.comp.ime.eb.br/pos/arquivos/publicacoes/dissertacoes/2015/2015-Reinaldo.pdf>. Acesso em: 10 mai. de 2019.
- 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.
- 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.

- ROCCA, B. 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.
- SAEED, I. A.; SELAMAT, A. ; ABUAGOUB, A. M. A. A survey on malware and malware detection systems. **International Journal of Computer Applications**, v. 67, p. 25–31, 2013.
- 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.
- SIKORSKI, M.; HOGIN, A. **Practical Malware Analysis**. 5. ed. [S.l.]: No Starch Press, 2012. 2-2 p.
- SUNG, A. H.; MUKKAMALA, S. ; XU, J. Static analyzer of vicious executables. **Proceedings - Annual Computer Security Applications Conference, ACSAC**, v. 0, p. 326–334, 2005.
- 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.
- YUXIN, D.; SIYI, Z. Malware detection based on deep learning algorithm. **Neural Computing and Applications**, v. 31, n. 2, p. 461–472, 2019.