UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS DE QUIXADÁ
CURSO DE GRADUAÇÃO EM CIÊNCIAS DA COMPUTAÇÃO

JOÃO MARCELO RAVACHE FERNANDES DA SILVA

CLASSIFICADOR DE MALWARES EM APLICATIVOS ANDROID ATRAVÉS DE REDES NEURAIS RECORRENTES

QUIXADÁ
2022

JOÃO MARCELO RAVACHE FERNANDES DA SILVA

CLASSIFICADOR DE MALWARES EM APLICATIVOS ANDROID ATRAVÉS DE REDES NEURAIS RECORRENTES

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciências da Computação do CAMPUS DE QUIXADÁ da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Ciências da Computação.

Orientador: Prof. Me. Roberto Cabral Rabêlo Filho.

QUIXADÁ

2022

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

S58c   Silva, João Marcelo Ravache Fernandes da.
Classificador de malwares em aplicativos android através de redes neurais recorrentes / João Marcelo Ravache Fernandes da Silva. – 2022.
36 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Quixadá, Curso de Ciência da Computação, Quixadá, 2022.
Orientação: Prof. Me. Roberto Cabral Rabêlo Filho.

1. Rede Neural Recorrente. 2. Malware. 3. Android (Programa de computador). I. Título.
CDD 004

JOÃO MARCELO RAVACHE FERNANDES DA SILVA

CLASSIFICADOR DE MALWARES EM APLICATIVOS ANDROID ATRAVÉS DE REDES NEURAIS RECORRENTES

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciências da Computação do CAMPUS DE QUIXADÁ da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Ciências da Computação.

Aprovada em: ___/___/_______.

BANCA EXAMINADORA

____

Prof. Me. Roberto Cabral Rabêlo Filho (Orientador)
Universidade Federal do Ceará (UFC)

____

Prof. Dr. Paulo de Tarso Guerra Oliveira
Universidade Federal do Ceará (UFC)

____

Prof. Me. Leodécio Braz da Silva Segundo
Analista de dados Minerva S.A

A Deus.

Aos meus pais, José Marcelo da Silva Fernandes
e Maria Socorro da Silva.

AGRADECIMENTOS

A UFC, pelo apoio e ensino, assim me ajudando no desenvolvimento profissional e pessoal.

Ao Prof. Roberto Cabral Rabêlo Filho, pela excelente orientação.

Aos professores participantes da banca examinadora Leodécio Braz da Silva Segundo e Paulo de Tarso Guerra Oliveira pelo tempo, pelas valiosas colaborações e sugestões.

Aos colegas da turma de graduação, pelas reflexões, sugestões recebidas, momentos gratificantes e amizade. Aos meus pais, que sempre me apoiam e que foi minha base para esta caminhada.

Aos meus familiares, que sempre me apoiaram e sempre me motivaram nos meus objetivos.

Ao meu amigo Marcelo, que esteve comigo nessa caminhada, nas conquistas, nos momentos bons e ruins.

A minha esposa Luciana dos Santos Feitosa, que sempre me deu força nos momentos difíceis, bons e entre outros, vem sendo minha motivação e minha base nessa caminhada.

Ao meu filho, que me deu forças para continuar em frente mesmo em tempos difíceis.

"Ainda que eu andasse pelo vale da sombra da morte, não temeria mal algum, porque tu estás comigo."
(Salmos 23:4)

RESUMO

Em 2019, o número de ataques de *malwares* aos dispositivos Android aumentou significativamente, devido ao crescimento da quantidade de aparelhos Android em uso no mundo. Estima-se que o número de ataques dobrou em relação a 2018. Para lidar com esse problema, foi desenvolvido um modelo de rede neural recorrente LSTM, que usa as chamadas de API dos aplicativos como entrada para classificar possíveis *malwares*. Os resultados mostraram que o modelo obteve um ótimo resultado para a abordagem de análise estática com as chamadas de API, onde obteve-se uma acurácia e precisão de 96,45% e 97,63%, respectivamente.

Palavras-chave: Rede Neural Recorrente; Malware; Android.

ABSTRACT

In 2019, the number of malware attacks on Android devices increased significantly due to the growth in the number of Android devices in use worldwide. It is estimated that the number of attacks has doubled compared to 2018. To deal with this problem, an LSTM recurrent neural network model was developed, which uses API calls from applications as input to classify possible malware. The results showed that the model obtained an excellent result for the static analysis approach with API calls, achieving accuracy and a precision of 96,45% e 97,63%, respectively.

Keywords: Recurrent Neural Network; Malware; Android.

LISTA DE FIGURAS

Figura 1 – Arquitetura Android. 14
Figura 2 – Representação de um neurônio artificial. 17
Figura 3 – À esquerda temos um neurônio biológico e à direita um neurônio Perceptron. 17
Figura 4 – Camadas RNN. 18
Figura 5 – Camadas internas de uma LSTM. 18
Figura 6 – Diagrama das fases do trabalho. 23
Figura 7 – Pré-processamento das amostras. 24
Figura 8 – O gráfico em azul é a loss do treinamento e o gráfico laranja é a loss da validação. 27
Figura 9 – O gráfico em azul é a loss do treinamento e o gráfico laranja é a loss da validação. 27
Figura 10 – O gráfico em azul é a loss do treinamento e o gráfico laranja é a loss da validação. 28
Figura 11 – O gráfico em azul é a loss do treinamento e o gráfico laranja é a loss da validação. 28
Figura 12 – O gráfico em azul é a loss do treinamento e o gráfico laranja é a loss da validação. 29
Figura 13 – O gráfico em azul é a loss do treinamento e o gráfico laranja é a loss da validação. 29

LISTA DE TABELAS

Tabela 1 – Trabalhos Relacionados 21
Tabela 2 – Parâmetros do modelo. 26
Tabela 3 – Resultados dos testes de 110 épocas, 165 épocas e 185 épocas. 30
Tabela 4 – Comparação dos resultados. 30

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
ART	Android Runtime
HAL	Camada de Abstração de Hardware
LSTM	*Long Short-Term Memory*
PLN	Processamento de Linguagem Natural
RNN	*Recurrent Neural Network*
SO	Sistemas Operacionais

SUMÁRIO

1 INTRODUÇÃO 12

2 FUNDAMENTAÇÃO TEÓRICA 14

2.1 Android 14

2.2 Malware 15

2.3 Técnicas de Análise de *malware* 15

2.3.1 Análise Estática 15

2.3.2 Análise Dinâmica 16

2.4 Redes Neurais Artificiais 16

2.5 Redes Neurais Recorrente 17

3 TRABALHOS RELACIONADOS 20

4 DESENVOLVIMENTO 22

4.1 Procedimentos Metodológicos 22

4.2 Selecionar a base de dados 22

4.3 Elaboração e Implementação do classificador 23

4.3.1 Extração das chamadas de API 23

4.3.1.1 Androguard *23*

4.3.2 Atividades de pré-processamento 23

4.3.3 Definição e treinamento do modelo 24

4.3.3.1 Tokenização *25*

4.3.3.2 Padding *25*

4.3.3.3 Embedding *25*

4.3.4 Modelo 25

4.4 Resultados dos treinamentos 26

5 CONCLUSÕES E TRABALHOS FUTUROS 31

REFERÊNCIAS 32

APÊNDICE A –CÓDIGOS-FONTES UTILIZADOS 34

1 INTRODUÇÃO

O uso de dispositivos móveis vem crescendo. Desde 2018 ocupam a maior fatia do mercado da computação e a tendência é que continuem crescendo nos próximos anos (CISCO, 2020). A quantidade de pessoas que utilizam dispositivos móveis já está em 4 bilhões, ocupando mais de 50% da população mundial (ANALYTICS, 2021). Os principais Sistemas Operacionais (SO) para dispositivos móveis, são o Android e IOS. O Android é o sistema operacional mais utilizado na maioria dos países, estando presente em 85,4% dos dispositivos e segundo a IDC (2020), continuará sendo o mais utilizado pelos próximos anos. No primeiro trimestre de 2020, segundo Clement (2020), a Google Play, principal loja de aplicativos Android, tornou-se a loja com mais aplicativos disponíveis, com cerca de 2,56 bilhões de aplicativos.

O sistema operacional Android usa um sistema de permissões, para controlar o acesso à recursos do dispositivo, o aplicativo solicita a permissão para acessar o armazenamento e recursos do sistema, através da *Application Programming Interface* (API) de privacidade e segurança do Android. Antes de instalar um aplicativo, o Android exige que sejam especificados quais recursos são requeridos pela aplicação; estas permissões estão localizadas em um arquivo de configuração e podem ser configuradas pelo desenvolvedor.

O sistema de permissões do Android permite que o usuário controle as permissões dos aplicativos, dando a ele a decisão de escolher o que o aplicativo pode ou não usar. Essas permissões vem evoluindo e aumentando a quantidade de recursos (WEI et al., 2012), mas na mesma proporção, aumentam as dificuldades de entendimento dos usuários sobre esses recursos, tornando-se um vetor de ataques para aplicativos maliciosos.

O crescimento do número de dispositivos Android aumentou a quantidade de *malwares* projetados para eles. Pesquisadores da Kaspersky Lab (RODRIGUES, 2019), descobriram que o número de ataques para dispositivos Android quase dobrou em um ano; chegando a 116,5 milhões de ataques em 2018, contra 66,4 milhões de ataques em 2017.

Com o crescente interesse em aplicações de aprendizado de máquina para classificar *malwares* em aplicativos Android entre pesquisadores de segurança, surgiu a dúvida sobre quais dados utilizar no treinamento (QIAO et al., 2016). Em aprendizagem de máquina, selecionar a base de dados para o treinamento é fundamental, pois além do modelo, os dados influenciam muito a precisão da classificação. Vários projetos de pesquisa utilizaram algoritmos de aprendizado de máquina a fim de classificar ou detectar *malwares* para Android; no trabalho de BAN et al. (2016) foram utilizadas as chamadas de API do Android e permissões, no trabalho de

MILOSEVIC et al. (2017) foram utilizadas as permissões e código-fonte, no trabalho de QIAO et al. (2016) foram utilizadas as chamadas de API do Android e permissões das aplicações maliciosas para treinar os modelos utilizados.

Este trabalho teve por objetivo implementar um classificador de *malware* com base nas sequência de chamadas de API. Para a realização desse objetivo foram seguidos os seguintes passos: selecionar a base de dados, desenvolver o modelo classificador, treinar o modelo e avaliar o classificador. Para alcançar estes passos foi utilizada a base de dados do *Canadian Institute for Cybersecurity* (MAHDAVIFAR et. al., 2020).

Neste trabalho, aplicou-se um modelo de *Recurrent Neural Network* (RNN), a fim de buscar um método para detecção de *malware*. Com o modelo pronto, realizou-se comparações com trabalhos da área no intuito de melhorar e mostrar que essa abordagem pode ser de grande valia na análise de *malware*. Através das características das sequências de chamadas de API e com aplicação do modelo *Long Short-Term Memory* (LSTM) observa-se um ótimo resultado na classificação de *malware* em aplicativos Android. O modelo foi criado utilizando a biblioteca do Keras, realizando o treinamento do modelo com base nas sequências de chamadas de API; a aplicação desse modelo no nosso escopo resultou, em média, uma acurácia de 96,45% e 97,63% de recall na classificação de *malwares* em aplicativos Android.

Os próximos capítulos estão organizados da seguinte maneira: no Capítulo 2, será apresentada a fundamentação teórica, onde há um levantamento de conceitos sobre *malwares*, conceitos de técnicas de análise de *malware*, Rede Neural, Rede Neural Recorrente e LSTM; no Capítulo 3, serão apresentados os trabalhos relacionados, com descrição e comparação de projetos e pesquisas que possuem aspectos similares com os propostos neste trabalho; no Capítulo 4, serão apresentados os procedimentos metodológicos, onde descrito o desenvolvimento deste trabalho; no Capítulo 5, serão apresentados o desenvolvimento e experimentos realizados com base na metodologia adotada, e seus resultados; no Capítulo 6, são mostradas as conclusões deste trabalho, bem como os trabalhos futuros.

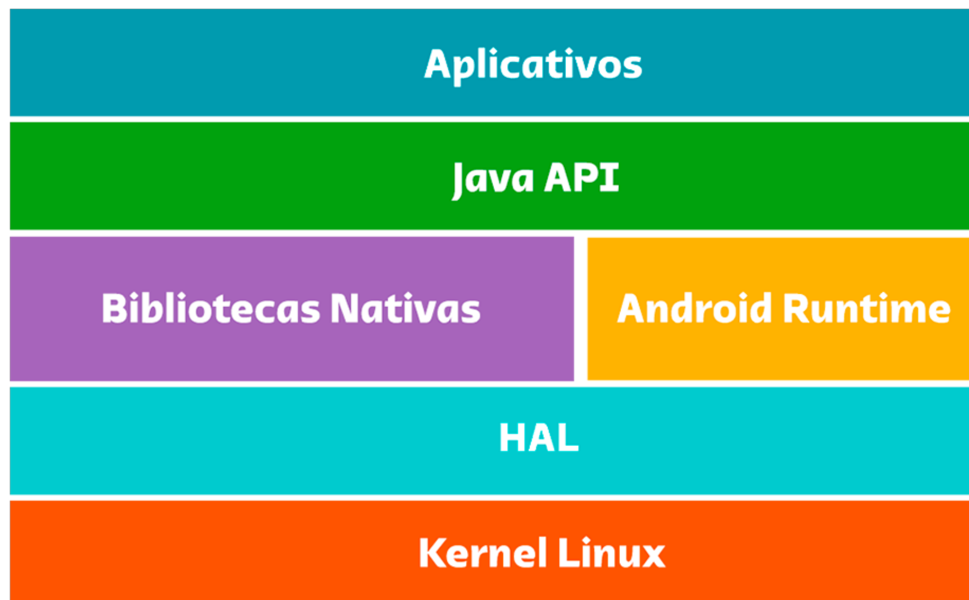
2 FUNDAMENTAÇÃO TEÓRICA

2.1 Android

O Android é baseado no kernel do Linux, que fornece recursos de segurança principais e permite que os fabricantes de dispositivos desenvolvam drivers de hardware para um kernel conhecido. O Android Runtime (ART) depende do kernel do Linux para cobrir algumas de suas funcionalidades (DEVELOPERS, 2020).

O android possui uma Camada de Abstração de Hardware (HAL) que fornece interfaces padrão, onde permite que recursos de hardware de maior nível. Já o ART é o ambiente de execução padrão para dispositivos Android, onde foi projetado para executar várias máquinas virtuais em dispositivos de baixa memória (DEVELOPERS, 2020).

Figura 1 – Arquitetura Android.



Fonte: Autor

O Android possui algumas bibliotecas nativas escritas em C/C++(linguagem de programação), onde algumas são acessadas via APIs, onde são as chamadas dessas APIs que são utilizadas como entrada para para o modelo LSTM, permitindo o acesso de aplicativos a essas bibliotecas. Por último na camada mais superficial do android, estão os aplicativos do android, onde possui os aplicativos já instalados por padrão (DEVELOPERS, 2020), como pode ver na Figura 1.

2.2 Malware

malware é qualquer tipo de programa malicioso que tem como objetivo prejudicar ou modificar o sistema de um dispositivo. Existem diversos tipos de *malwares*, cada um com seu próprio objetivo, desde roubo de dados, roubo de credenciais e até mesmo tornar o dispositivo inoperacional (GOOGLE PLAY PROTECT, 2020). Estes programas maliciosos podem ser distribuídos por meio de anexos em emails, downloads não autorizados ou distribuídos por meio de sites não oficiais.

2.3 Técnicas de Análise de *malware*

A análise de *malwares* utiliza métodos e ferramentas para dissecar *malwares*, a fim de entender melhor seu funcionamento com a finalidade de identificar suas características principais para classificá-los (SIKORSKI; HONIG, 2012). Nas seções a seguir será explicado o tipo de análise estática e análise dinâmica.

2.3.1 Análise Estática

Esta técnica descreve o processo de análise do código ou estrutura de um determinado aplicativo, com a finalidade de identificar e conhecer o funcionamento de suas funções sem executá-lo (SIKORSKI; HONIG, 2012). A análise estática pode ajudar a entender as técnicas, pontos fortes, fracos e limitações (OWASP, 2019).

Durante a análise estática, o código-fonte do aplicativo é analisado para garantir que a implementação do aplicativo não contenha códigos nocivos. Na maioria dos casos envolvendo análise estática, é usada uma abordagem automática e manual. As análises automáticas são scanners, que buscam trechos no código-fonte mais frequentes de *malwares*, isto é, características mais rotineiras dos *malwares*, sendo as análises mais profundas realizadas de maneira manual pelo pesquisador (OWASP, 2019).

A análise estática começa com uma revisão manual do código-fonte do aplicativo em busca de comportamento prejudicial. Podem ser utilizados métodos de pesquisa por palavras-chaves com o comando grep do linux no código-fonte e IDEs (ambiente de desenvolvimento integrado) com funções básicas de revisão de código. Com isso, é feito uma análise linha por linha do código-fonte a fim de encontrar comportamentos que se julguem prejudiciais (OWASP, 2019).

Uma das ferramentas utilizadas neste trabalho para extração das sequências de chamadas de API foi o Androguard (DESNOS, 2019), que é uma ferramenta completa para manipulação de arquivos Android.

2.3.2 Análise Dinâmica

Esta técnica descreve o processo de análise do código em tempo de execução, visando obter informações em memória e obter comportamentos que possam ser caracterizados como um *malware*. Essa análise traz riscos, por isso, muitos pesquisadores criam um ambiente isolado e sem riscos, conhecido por sandbox (SIKORSKI; HONIG, 2012).

Existem algumas maneiras de analisar de forma dinâmica aplicativos android fazendo uso de ferramentas próprias para esse uso, uma delas é utilizar uma imagem modificada a fim de monitorar e extrair todas as operações em tempo de execução de um aplicativo. Uma dessas é o DroidBox que é voltado para o Android 4.1.2 e para dispositivo Nexus 4, onde a mesma monitora e extrai permissões burladas, hashes, dados da rede, SMS enviados etc.

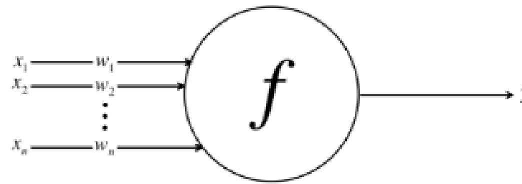
2.4 Redes Neurais Artificiais

As redes neurais artificiais (RNAs) são ferramentas de modelagem computacional muito utilizadas e aceitas para solucionar problemas complexos do mundo real. As Redes Neurais Artificiais são construídas com base em modelos matemáticos que simulam o comportamento dos neurônios biológicos. Estes modelos são usados para criar redes neurais artificiais que podem ser usadas para resolver problemas complexos. Estas redes são capazes de aprender a partir de dados, ajustando seus parâmetros internos para produzir resultados desejados (BUDUMA; LOCASCIO, 2017). Elas também podem ser treinadas para realizar tarefas específicas, como reconhecimento de padrões e classificação de imagens.

A analogia entre neurônios biológicos e artificiais está nas conexões entre os nós que representam os axônios e dendritos, as conexões com peso representam as sinapses e a soma se aproxima ao limiar de um neurônio biológico.

Com isso, podemos aplicar esse conhecimento do modelo biológico para o modelo computacional como pode ser visto na Figura 2, cada neurônio perceptron recebe uma sequência de entradas x1, x2, ..., xn, onde cada entrada recebe um peso w1, w2, . . . , wn. Na Figura 3, podemos ver que cada entrada é multiplicada pelo seu respectivo peso, por fim, todos são

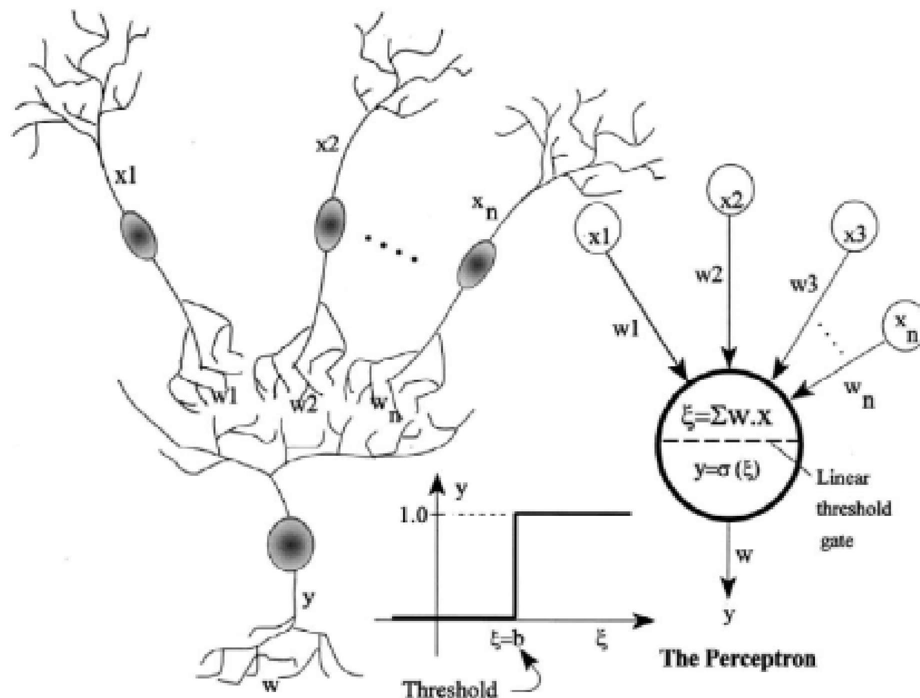
Figura 2 – Representação de um neurônio artificial.



Fonte: BUDUMA; LOCASCIO, 2017

somados e aplicadas ao bias(parâmetro para ajustar a saída da somatória), assim passando para uma função de ativação que determina quais neurônios serão ativados e quais não serão.

Figura 3 – À esquerda temos um neurônio biológico e à direita um neurônio Perceptron.



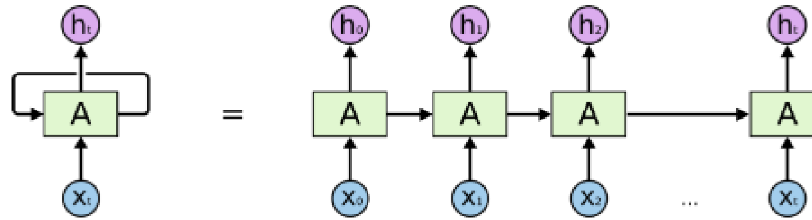
Fonte: BASHEER, 2000

2.5 Redes Neurais Recorrente

Redes Neurais Recorrentes (RNN) são redes sequenciais, com isso elas conseguem guardar ou persistir informações. A Figura 4, representa bem como a rede neural recorrente funciona, uma entrada Xt e uma saída ht, onde a saída gerada pela rede, temos um loop que permite que informações anteriores persistam; com isso, a informações podem ser passadas de uma etapa para outra. Na mesma Figura 4, pode-se perceber que, na verdade, o modelo é cópia

de si mesmo, composto por camadas que passam informações sequenciais.

Figura 4 – Camadas RNN.

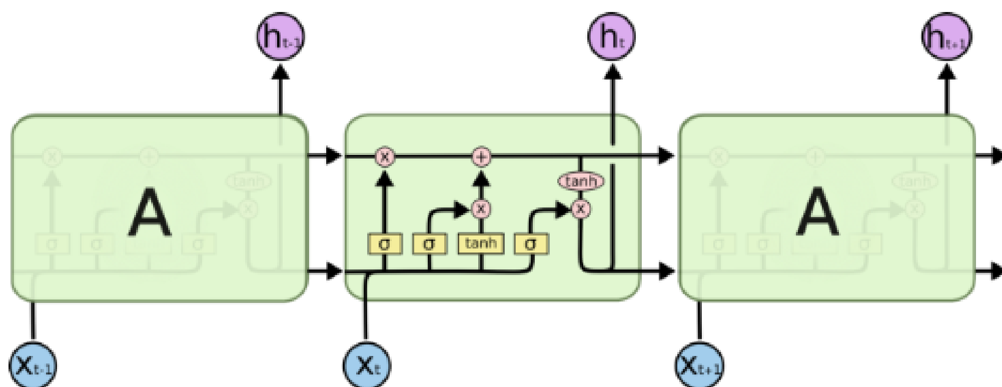


Fonte: OLAH, 2015

A rede neural recorrente tem um problema que foi bastante explorado por Hochreiter (1991); à medida que as informações passam por suas camadas, algumas informações são perdidas, fazendo com que entradas não tão recentes não sejam lembradas à medida que o modelo é treinado, assim algumas informações importantes são removidas do modelo durante o treinamento.

A LSTM surgiu como solução para esse problema(HOCHREITER, 1997), mas as redes LSTM tem uma estrutura diferente da RNN padrão, onde cada uma das camadas possui várias outras camadas, como pode ser visto na Figura 5. A LSTM possui uma linha transportadora que é o estado da célula, a qual consegue remover ou adicionar informações ao estado da célula.

Figura 5 – Camadas internas de uma LSTM.



Fonte: OLAH, 2015

Essas operações são reguladas pelos “portões” e são uma forma de deixar as informações passarem, através de uma função de ativação sigmóide, onde 0 significa fechado e 1 significa aberto, assim fazendo com que o dado passe ou não. Assim, possuindo 3 portões, um portão de entrada, onde será decidido o que atualizar do estado da célula a partir da nova entrada. O portão de saída que a partir do estado da célula, ele filtra e entrega uma saída. O portão do

esquecimento, que decide o que será apagado de cada valor do estado da célula, como pode ser vista na Figura 5.

3 TRABALHOS RELACIONADOS

A utilização de algoritmos de aprendizado de máquina para a detecção e classificação de *malwares* para Android ainda se encontra em fase de evolução. Os trabalhos a seguir buscam propor novas técnicas e modelos para melhorar a detecção de *malwares* utilizando aprendizado de máquina.

No trabalho de BAN et al. (2016), foi criado um sistema de análise e detecção, onde utiliza-se extração de chamadas de API, permissões, categorias e descrições de características, de forma estática. Dado a extração, foram passados para o classificador e mesclado as entradas, utilizando o algoritmo SVM (Support Vector Machine). Foi obtido 94,07% de acurácia e 87,23% de precisão utilizando como entrada as chamadas de API. Assim como no trabalho citado, este trabalho utilizou extração de chamadas de API estática para treinar o classificador LSTM.

No trabalho de QIAO et al. (2016), foi criado um sistema de detecção onde utiliza-se da mesclagem de permissões e recursos de API para detectar *malwares*; eles utilizaram de três algoritmos RandomForest, Rede Neural e SVM (Support Vector Machine). Por fim, com o treinamento e avaliação de cada um dos modelos através de análises qualitativas e quantitativas, através das chamadas de API o algoritmo que teve o melhor resultado foi o de Rede Neural com 93,48% de acurácia. Assim como no trabalho citado, este trabalho utilizou rede neural recorrente como classificador.

No trabalho de MILOSEVIC et al. (2017), foram utilizadas duas abordagens de análise estática. Na primeira foram analisadas as permissões que o aplicativo solicita e na segunda foi analisado o código-fonte do aplicativo. Após isso, foram utilizadas duas formas de aprendizado de máquina, isto é, classificação e clustering, onde foram utilizados os algoritmos SVM (Support Vector Machine), Naive Bayes, árvores de decisão e JRIP ou Ripper (Repeated Incremental Pruning to Produce Error Reduction) para análise das permissões. Já para análise de código-fonte, foram utilizados árvores de decisão, Naive Bayes, vetor de suporte com otimização sequencial mínima, RandomForest, JRIP, regressão logística e SVM. Com isso, foi avaliado a precisão de cada algoritmo aplicado; os algoritmos baseados em SVM se saíram melhor, com o algoritmo SVM sendo o melhor, onde foi obtido 85,2% de precisão. Assim como no trabalho citado, este trabalho utilizou extração da sequência das chamadas de API do código-fonte.

Na Tabela 1 é possível ver a comparação entre os estudos apresentados e o nosso trabalho. Como pode ser observado, nosso modelo apresenta uma abordagem diferente; visto que utilizamos rede neural recorrente e como entrada para o modelo as sequências de chamadas

de API, mas com uma base de dados diferente dos outros estudos. Esta abordagem foi escolhida devido à sua capacidade de lidar com dados sequenciais, que são fornecidos pelas chamadas de API.

Tabela 1 – Trabalhos Relacionados

Estudo	Algoritmos	Dados utilizados para o treinamento
Ban et al.	SVM	Chamadas de API
Qiao et al.	Rede Neural	Permissões e chamadas de API1
Milosevic et al.	SVM	Chamadas de API do código fonte
Presente Trabalho	Rede Neural Recorrente LSTM	Sequência das Chamadas de API

Fonte: Autor.

4 DESENVOLVIMENTO

4.1 Procedimentos Metodológicos

O trabalho foi desenvolvido através da elaboração, desenvolvimento e teste de um modelo classificador de *malwares* com base na extração das sequências das chamadas de API no Android Package. A partir da base de dados disponibilizada pelo *Canadian Institute for Cybersecurity* (MAHDAVIFAR et. al., 2020) foi possível extrair as sequências das chamadas de API, a fim de utilizá-las no classificador.

A base de dados possui 4036 amostras benignas e 4362 de *malware* (mas foi utilizado somente 4036 amostras para equilibrar as amostras). Logo, foi implementado um modelo de Rede Neural Recorrente LSTM e a implementação do modelo classificador elaborado, utilizando o TensorFlow (ABADI et al., 2015) que é uma biblioteca de código aberto para aprendizado de máquina, juntamente com o Keras (CHOLLET, 2015) que é uma biblioteca de rede neural que roda sobre o TensorFlow (ABADI et al., 2015), para a implementação do modelo.

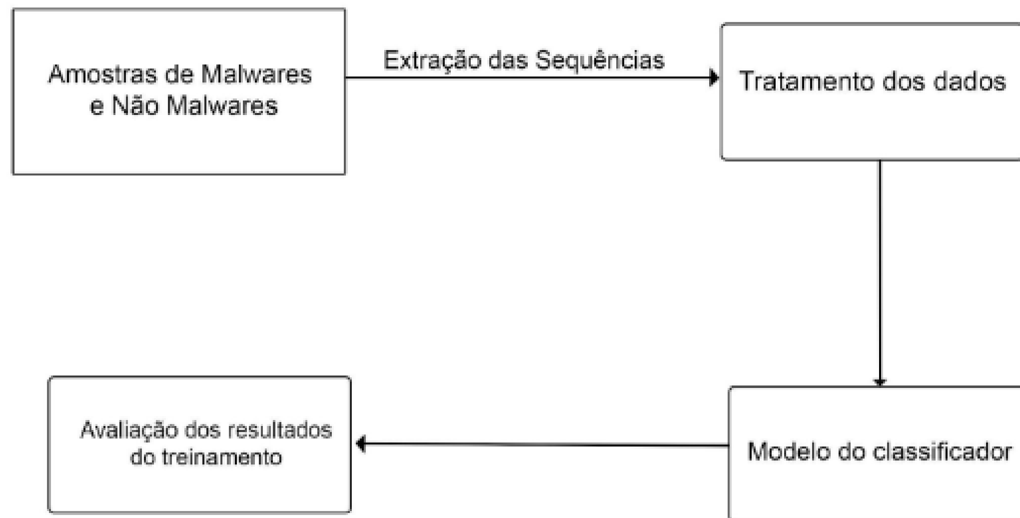
Para o treinamento, foi utilizado o Google Colab que é um serviço na nuvem para a realização e incentivo de pesquisas em Aprendizado de Máquina e Inteligência Artificial, permitindo escrever código python no navegador com GPUs gratuitas e sem necessidade de configuração inicial. Com isso, foram analisados os resultados obtidos e realizada uma comparação com resultados já existentes.

4.2 Selecionar a base de dados

A base de dados foi escolhida pela grande quantidade de amostras e por ter tanto amostras benignas como *malwares*. Com isso, o trabalho foi dividido em quatro fases: (1) extração e tratamento de dados a partir da base de dados escolhida contendo apks de *malware* e não *malware*; (2) implementação e treinamento do classificador utilizando LSTM, gerando um modelo de classificação de chamadas de API; (3) validação do modelo LSTM de classificação de chamadas de API no conjunto de teste; e (4) avaliação dos resultados do modelo, como pode ser visto na Figura 6.

Cada amostra é composta por um aplicativo do tipo apk, onde a extração das sequências de chamadas de API foi realizada com auxílio da biblioteca Androguard (DESNOS, 2019), que nos forneceu sequências que servirão de entrada para o treinamento do modelo.

Figura 6 – Diagrama das fases do trabalho.



Fonte: Autor.

4.3 Elaboração e Implementação do classificador

4.3.1 Extração das chamadas de API

Para extrair as chamadas de API foi desenvolvido um script para fornecer as sequências de todas as chamadas de API dos arquivos apk com o auxílio da biblioteca Androguard (DESNOS, 2019).

4.3.1.1 Androguard

Para realizar uma análise estática com o adroguard é preciso utilizar o método AnalyseAPK da biblioteca do androguard. O AnalyseAPK é um método que retorna um objeto APK, uma lista de DalvikVMFormat e um objeto Analysis. A lista retornada de objetos DalvikVMFormat é uma lista que contém análises de todas as sequências de classes utilizadas pelo apk.

4.3.2 Atividades de pré-processamento

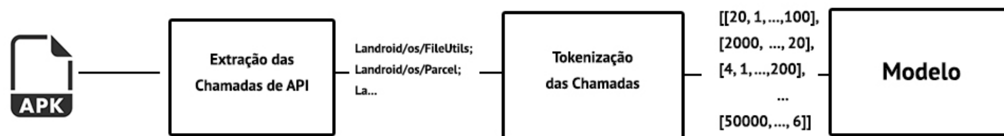
A atividade de pré-processamento foi a partir da extração das chamadas de API de cada apk, onde foram organizados em arquivos do tipo calls, onde em cada arquivo continha a sequência de chamadas de API separadas por ";". Um exemplo de sequência getPackageName;<init>;get;remove;put;<init>;getExtras e assim por diante. O código utilizado para extração, nos permite percorrer toda a lista DalvikVMFormat e guardar as chamadas de API em

uma lista de métodos, assim foi possível extrair todas as chamadas de API do Android, como pode ser visto no Código do Apendice A.

As amostras foram separadas em três partes de forma estratificada, onde em ambas as partes as amostras de *malware* e benigno estão distribuídas em quantidades iguais, com isso ficou uma para o treinamento, contendo 60% das amostras, outra para validação contendo 20% das amostras e outra para teste com 20% das amostras. Assim, das 8072 amostras, 4036 amostras são benignas e 4036 amostras são *malwares*.

Cada conjunto foi separado em 4844 para o conjunto de treinamento, 1614 para validação e 1614 para o conjunto de testes. As amostras de treino foram utilizadas para treinar o modelo LSTM de classificação, já a validação foi usada para comparar o modelo de treinamento durante o treino, assim utilizando as amostras de testes para obter o resultado final da classificação do modelo.

Figura 7 – Pré-processamento das amostras.



Fonte: Autor.

Na Figura 7, é possível ver o processamento dos dados antes de chegar ao modelo, onde o apk entra no módulo de extração e retorna a sequência de chamadas de API desse aplicativo, com essa sequência é passado para o módulo de tokenização, onde essas sequências de chamadas de API são transformadas em tokens, a partir dessa transformação, a sequência em tokens é passado para o modelo.

Após a extração ocorreu o problema com o tamanho da sequência, onde no treinamento do modelo, as épocas ficaram com os tempos muito grandes pela limitação de hardware da máquina que estava ao alcance. Com isso foi selecionado somente chamadas disponibilizadas pela api do android.

4.3.3 Definição e treinamento do modelo

Para representar as sequências para o treinamento foi utilizada a classe Tokenizer do Keras (CHOLLET, 2015) que permite vetorizar um texto, onde ele aplica essa vetorização em cada palavra de um texto. Essa vetorização gera uma sequência de inteiros, neste caso será

vetorizado cada chamada da sequência em inteiros. Com isso, foi calculada a quantidade de chamadas únicas que continham em todas as sequências, para que seja informada na classe Tokenizer e gerar a entrada para a camada de Embedding do Keras, que irá fornecer uma representação densa de palavras e seus significados relativos (MIKOLOV et al., 2013).

4.3.3.1 Tokenização

Esse processo permite que seja possível transformar uma sequência de palavras em tokens, ou seja, será criado um vetor com a representação da frase em números, assim facilitando a manipulação dos dados pelo modelo.

4.3.3.2 Padding

É preciso ter entradas de tamanhos fixos e nem todas as sequências possuem o mesmo tamanho, para resolver isso é preciso aplicar um ajuste nas sequências definindo um tamanho máximo, assim é ajustado todas as sequências para esse tamanho e as sequências menores que o tamanho fixo é preenchidas com zero.

4.3.3.3 Embedding

É uma camada de incorporação do Keras, onde retorna a vetorização do vocabulário de palavras em forma de valores reais, fazendo o uso desse vetor para dar sentido a cada palavra dentro de uma frase.

4.3.4 Modelo

O modelo foi criado com 200 unidades de memória LSTM, que no caso é o units que é um componente básico do modelo LSTM, responsável por armazenar informações ao longo do tempo. Com isso, foi utilizado um batch size de tamanho 64, após testes com diferentes valores de batch size, o modelo se saiu melhor. Por fim, a input_dim é o tamanho do vocabulário, ou seja, a quantidade de palavras diferentes que existe nas amostras, esse valor foi escolhido através de um varredura a procura de todas as chamadas únicas com todas as amostras e o input_length foi escolhido através de um varredura a procura da maior sequência entre as amostras, assim o modelo foi construído, como pode ser visto na Tabela 2.

Tabela 2 – Parâmetros do modelo.

Parâmetros do modelo	Valores
$\text{input}_d im$	500
$\text{output}_d im$	128
$\text{input}_l ength$	6000
units	200
$\text{batch}_s ize$	64

Fonte: Autor.

4.4 Resultados dos treinamentos

Conforme mencionado anteriormente, o conjunto foi separado em 60% das amostras para treinamento, outra para validação contendo 20% das amostras e outra para teste com 20% das amostras, realizando pequenos treinamentos de 200 épocas. Os primeiros testes foram feitos em uma máquina com um processador Intel(R) Core(TM) i5-9300H, mas por conta da limitação de núcleos e problemas com longo treinamento, os testes finais foram feitos no Google Colab Pro onde possui GPU Tesla K80.co 2496 CUDA cores e 12 GB de memória, foi visto como uma ótima alternativa para realizar o treinamento.

Foram realizados testes com batch size de tamanho 64 e 32 com 200 épocas e subdivididos entre sem dropout, com dropout de 10% e com dropout de 20%, a fim de encontrar um modelo que apresente o melhor resultado entre estas variações de modelos, os testes seguiram a seguinte ordem: batch size de tamanho 64 e com dropout de 20%, batch size de tamanho 64 e com dropout de 20%, batch size de tamanho 64 e com dropout de 10%, batch size de tamanho 64 e sem dropout, batch size de tamanho 32 e com dropout de 20%, batch size de tamanho 32 e com dropout de 10% e batch size de tamanho 32 e sem dropout.

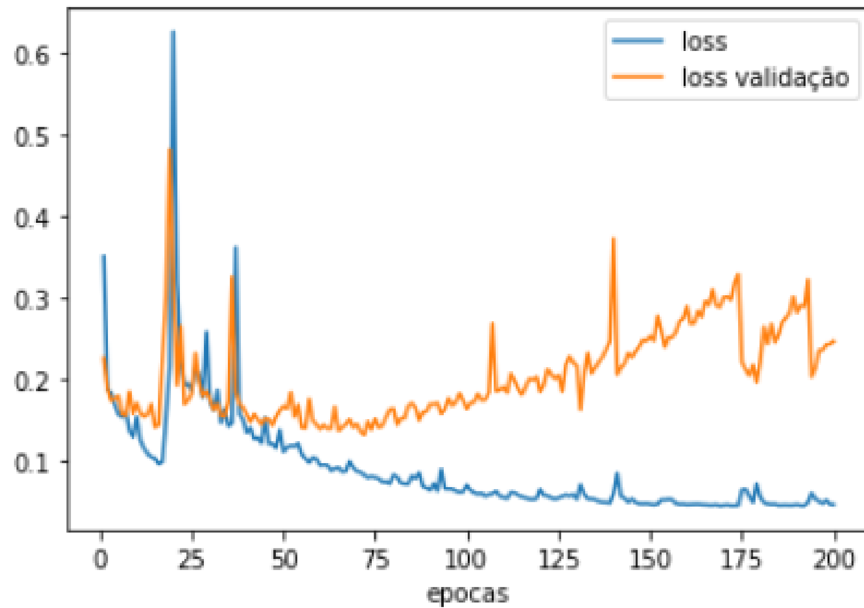
O teste feito com 64 de batch size e 200 épocas e dropout de 20%, o modelo se saiu muito bem com acurácia de 97% e um Recall de 97,8%, mas acabou ocorrendo overfitting que é possível ver no gráfico de Loss, como pode ser visto na Figura 8.

O teste feito com 64 de batch size e 200 épocas e dropout de 10%, o modelo não melhorou e acabou ficando com acurácia de 96% e um Recall de 95,7%, mas acabou ocorrendo também overfitting que é possível ver no gráfico de Loss, como pode ser visto na Figura 9.

O teste feito com batch size de tamanho 64 e 200 épocas e sem dropout, o modelo melhorou e acabou ficando com acurácia de 98% e um Recall de 98,3%, mas acabou ocorrendo também overfitting que é possível ver no gráfico de Loss, como pode ser visto na Figura 10.

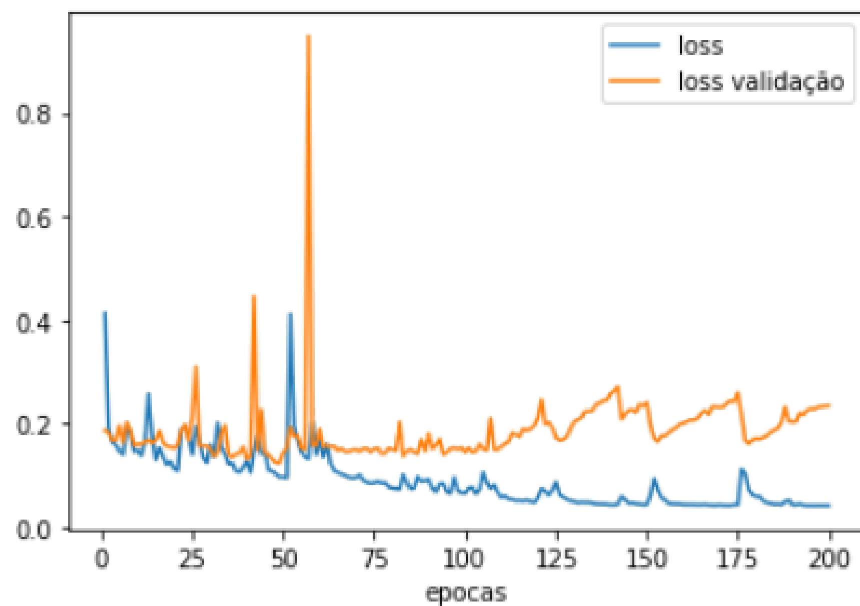
O teste feito com batch size de tamanho 32 e 200 épocas e dropout de 20%, o modelo

Figura 8 – O gráfico em azul é a loss do treinamento e o gráfico laranja é a loss da validação.



Fonte: Autor.

Figura 9 – O gráfico em azul é a loss do treinamento e o gráfico laranja é a loss da validação.

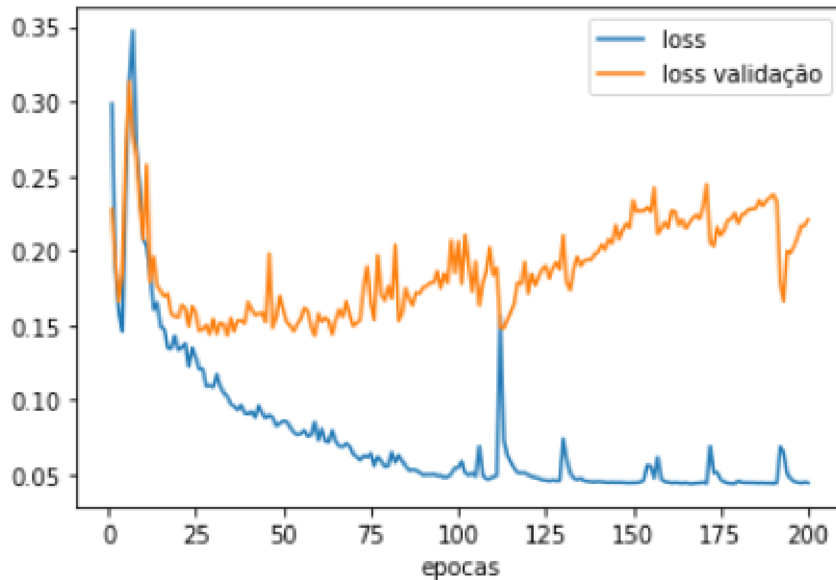


Fonte: Autor.

se saiu muito bem, mas mostrou resultados um pouco pior do que o de 64 de batch size, onde apresentou acurácia de 96% e um Recall de 96,7%, mas acabou ocorrendo overfitting até maior que os testes anteriores que é possível ver no gráfico de Loss, como pode ser visto na Figura 11.

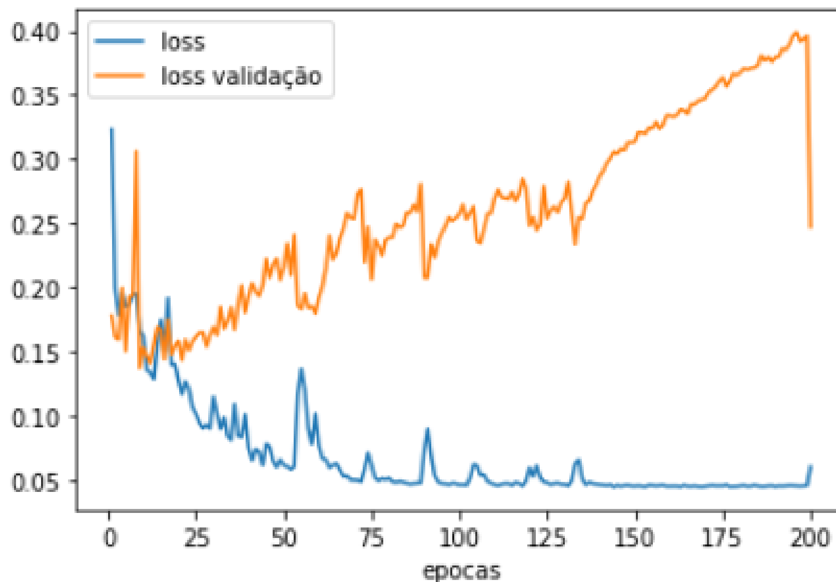
O teste feito com batch size de tamanho 32 e 200 épocas e dropout de 10%, o modelo se acabou apresentando uma acurácia da do teste anterior de 96%, mas com um Recall de 96,4%,

Figura 10 – O gráfico em azul é a loss do treinamento e o gráfico laranja é a loss da validação.



Fonte: Autor.

Figura 11 – O gráfico em azul é a loss do treinamento e o gráfico laranja é a loss da validação.

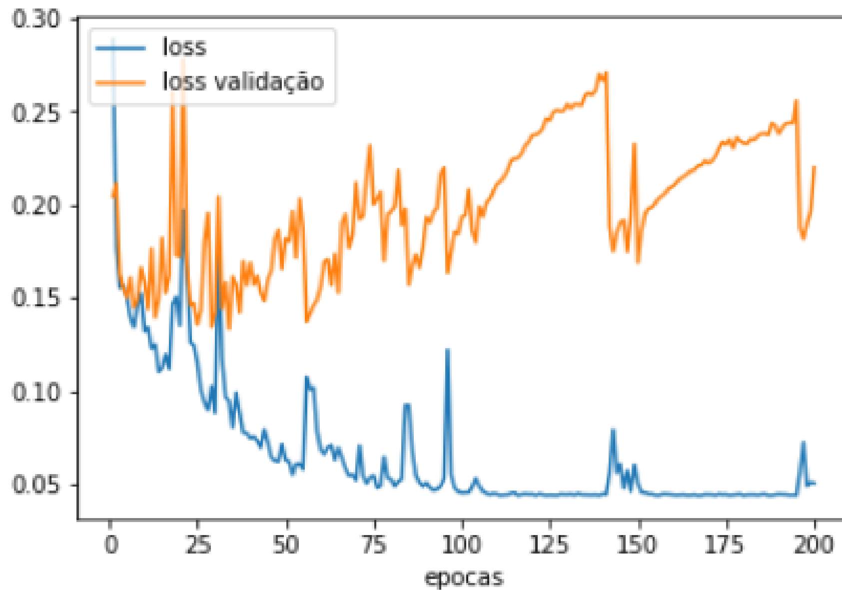


Fonte: Autor.

o overfitting acabou sendo até menor do que o anterior, mas semelhante aos testes com 64 de batch size que é possível ver no gráfico de Loss, como pode ser visto na Figura 12.

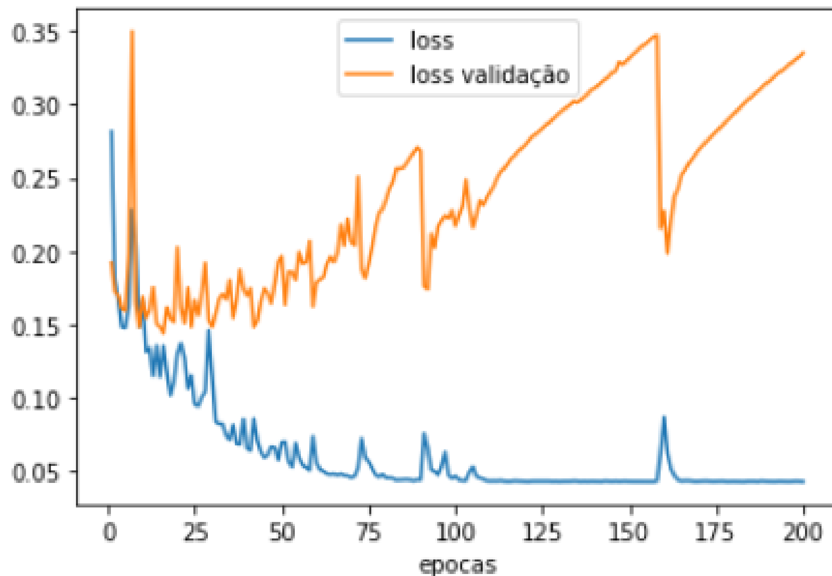
O teste feito com batch size de tamanho 32 e 200 épocas e sem dropout, o modelo acabou apresentando uma acurácia melhor que os testes com dropout obtendo um resultado de 97% e um Recall de 96,9%, acabou tendo overfitting também, é possível ver no gráfico de Loss, como pode ser visto na Figura 13.

Figura 12 – O gráfico em azul é a loss do treinamento e o gráfico laranja é a loss da validação.



Fonte: Autor.

Figura 13 – O gráfico em azul é a loss do treinamento e o gráfico laranja é a loss da validação.



Fonte: Autor.

Foi selecionado o batch size de tamanho 64 e sem dropout, onde os resultado apresentados foram os melhores, a partir dessa estrutura do modelo, foram realizados mais testes a fim de melhorar o modelo e os resultados, no caso foi feito uma redução de épocas para buscar um valor ideal de épocas para o modelo em que a loss tanto do treinamento como da validação fossem baixas e assim buscar um modelo sem tanto overfitting.

Com isso foram realizados 5 testes em cada número de épocas onde a loss do modelo

Tabela 3 – Resultados dos testes de 110 épocas, 165 épocas e 185 épocas.

Quantidade de épocas	Média recall	Desvio padrão
110	96,97%	0,73
165	97,63%	0,59
185	97,27%	1,14

Fonte: Autor.

de validação estiveram baixas, no caso foram utilizados 110 épocas, 165 épocas e 185 épocas, todos escolhidos de acordo com a loss do treino e a loss de validação, buscando uma loss baixa entre as duas loss, como pode ser visto na Tabela 3.

Após os testes, é visível que com 165 épocas o modelo obteve um ótimo resultado, onde a média dos resultados foi de 97,63% e a dispersão dos resultados em relação a média dos resultados não foi alta, em comparação com os outros resultados, o modelo de 165 épocas obteve resultados mais representativos.

Tabela 4 – Comparação dos resultados.

Estudo	Melhores Resultados de acurácia	Melhores Resultados de precisão	Melhores Resultados de recall
Ban et al.	94,07%	x	x
Qiao et al.	93,48%	x	x
Milosevic et al.	x	95,2%	x
Presente Trabalho	96,45%	95,28%	97,63%

Fonte: Autor.

Assim como pode ser visto na Tabela 4, o presente trabalho apresenta uma ótima acurácia e recall, onde o recall nos dá uma porcentagem da assertividade do modelo em relação a classificação de *malwares*. Diante dos resultados apresentados pelos trabalhos que utilizam como entrada chamadas de API ou código-fonte, mas com uma base de dados diferente deste trabalho, o modelo ainda se mostrou ser uma ótima ferramenta para classificação de *malware*.

5 CONCLUSÕES E TRABALHOS FUTUROS

Neste trabalho foi proposto um modelo classificador de *malware* através de chamadas de APIs, incentivado pela dificuldade em classificar uma grande quantidade de *malware*. Onde se mostrou ser muito preciso, assim podendo afirmar ser um grande agente de auxílio na classificação de *malwares* de aplicativos Android.

Uma das grandes dificuldades encontradas foi a restrição de hardware na máquina pessoal. Para resolver esse problema, migramos para o colab. Mesmo utilizando colab, tivemos problemas por conta do tamanho das sequências. Por fim, optamos pelo uso do colab pro, permitindo assim realizar treinamentos mais longos e aproveitando o máximo das amostras encontradas.

Ao usar as chamadas de APIs do Android no modelo, tivemos resultados extremamente satisfatórios. O classificador foi bastante eficaz ao classificar *malwares*. A acurácia média atingida foi de 96,45%, enquanto o recall chegou a 97,63%.

Alguns trabalhos futuros que podem ser realizados a partir deste estudo são:

- Explorar o uso de modelos LSTM com chamadas de API dinâmicas para melhorar a classificação de *malwares*;
- Investigar outros tipos de modelos de Rede Neural Recorrente;
- Criar um modelo que utilize a saída dos outros modelos para obter o melhor resultado possível;
- Explorar o uso de transformers que são modelos de aprendizado de máquina baseados em Processamento de Linguagem Natural (PLN).

REFERÊNCIAS

ABADI, M. e. a. Tensorflow: A system for large-scale machine learning. **12th USENIX symposium on operating systems design and implementation (OSDI 16)**, p. 265–283, 2016.

ANALYTICS, S. **Strategy Analytics: Half the World Owns a Smartphone**. 2021. Disponível em: https://news.strategyanalytics.com/press-releases/press-release-details/2021/Strategy-Analytics-Half-the-World-Owns-a-Smartphone/default.aspx. Acesso em: 10 Dez. de 2022.

BAN T. BAN, e. a. Integration of multi-modal features for android malware detection using linear svm. **2016 11th Asia Joint Conference on Information Security (AsiaJCIS), Fukuoka, Japan**, p. 141–146, 2016.

BAN T. BAN, e. a. Fundamentals of deep learning: Designing nextgeneration machine intelligence algorithms. **1st. ed. [S.l.]: O'Reilly Media, Inc.**, 2017.

BASHEER I.A BASHEER. HAJMEER, M. H. Artificial neural networks: fundamentals, computing, design, and application. **Journal of Microbiological Methods**, v. 43, p. 3–31, 2000.

CHOLLET, F. **keras**. 2015. Disponível em: https://github.com/fchollet/keras. Acesso em: 01 de Abr. de 2021.

CISCO. **Cisco Annual Internet Report (2018–2023)**. 2020. Disponível em: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html. Acesso em: 06 de Ago. de 2020.

CLEMENT., J. **Number of apps available in leading app stores as of 1st quarter 2020**. 2020. Disponível em: https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores. Acesso em: 06 de Ago. de 2020.

DESNOS, A. **Androguard**. 2019. Disponível em: https://github.com/androguard/androguard/releases/tag/v3.3.5. Acesso em: 01 Abr. de 2021.

GOOGLE. **Arquitetura da plataforma**. 2020. Disponível em: https://developers.google.com/android/play-protect/phacategories. Acesso em: 30 de Abr. de 2021.

GOOGLE. **Malware categories**. 2020. Disponível em: https://developer.android.com/guide/platform?hl=pt-br. Acesso em: 20 de Dez. de 2022.

HOCHREITER, S. **Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München**. 1991.

HOCHREITER, S.; SCHMIDHUBER, J. Long short-term memory. **Neural computation**, v. 9, p. 1735–80, 12 1997.

IDC. **Smartphone Market Share**. 2020. Disponível em: https://www.idc.com/promo/smartphone-market-share/os. Acesso em: 06 de Ago. de 2020.

MAHDAVIFAR ANDI FITRIAH ABDUL KADIR, R. F. D. A. A. A. G. S. Dynamic android malware category classification using semi-supervised deep learning. **The 18th IEEE International Conference on Dependable, Autonomic, and Secure Computing (DASC), Aug. 17-24, 2020**, 2020.

MIKOLOV, T.; CHEN, K.; CORRADO, G.; DEAN, J. **Efficient estimation of word representations in vector space**. 2013. Disponível em: https://arxiv.org/pdf/1301.3781.pdf. Acesso em: 29 Set. 2019.

MILOSEVIC, N.; DEHGHANTANHA, A.; CHOO, K.-K. Machine learning aided android malware classification. **Computers Electrical Engineering, v. 61**, 2017.

OLAH, C. **Understanding LSTM Networks**. 2015. Disponível em: https://colah.github.io/posts/2015-08-Understanding-LSTMs. Acesso em: 28 Ago. de 2020.

OWASP. **Mobile Security Testing Guide - 1.1.3 Release**. 2019. Disponível em: https://leanpub.com/mobile-security-testing-guide. Acesso em: 01 Abr. de 2021.

QIAO, M.; SUNG A. H.AND LIU, Q. Merging permission and api features for android malware detection. **5th IIAI International Congress on Advanced Applied Informatics (IIAI-AAI), Kumamoto, 2016**, 2016.

RODRIGUES, R. **Ataques contra dispositivos móveis dobram em 2018**. 2019. Disponível em: https://www.kaspersky.com.br/blog/ataques-moveis-malware/11480. Acesso em: 06 de Ago. de 2020.

SIKORSKI MICHAEL; HONIG, A. Practical malware analysis: The hands-on guide to dissecting malicious software. **1ª Edição. Local de publicação: No Starch Press**, 2016.

WEI, X.; GOMEZ, L.; NEAMTIU, I.; FALOUTSOS, M. Permission evolution in the android ecosystem. **In Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC ’12). Association for Computing Machinery, New York, NY, USA**, p. 31–40, 2012.

APÊNDICE A – CÓDIGOS-FONTES UTILIZADOS

Código-fonte 1 – Código para extrair as chamadas de API

```
def EXTRACT_METHOD_CALLS_OF_API_ANDROID(pathApk):
    app, list_of_dex, dx = AnalyzeAPK(pathApk)

    methods_list = list()

    for method in dx.get_methods():
        for _, call, _ in method.get_xref_to():
            aux = (call.class_name).replace('$','')
            if( (Landroid/' in aux) ):
                methods_list.append(call.name+';')
    return methods_list
```