Universidade Federal de Ouro Preto
Escola de Minas
CECAU - Colegiado do Curso de
Engenharia de Controle e Automação

Wellington Resende de Araújo Júnior

Sistema de Monitoramento e Detecção de Ameaças em Redes Utilizando Aprendizado de Máquina

Monografia de Graduação

Ouro Preto, 2025

Wellington Resende de Araújo Júnior

Sistema de Monitoramento e Detecção de Ameaças em Redes Utilizando Aprendizado de Máquina

Trabalho apresentado ao Colegiado do Curso de Engenharia de Controle e Automação da Universidade Federal de Ouro Preto como parte dos requisitos para a obtenção do Grau de Engenheiro(a) de Controle e Automação.

Universidade Federal de Ouro Preto

Orientador: Gabriel Vinicios Moreira Fernandes, Me.

Coorientador: Prof. Agnaldo José da Rocha Reis, Dr.

Ouro Preto

2025

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE OURO PRETO
REITORIA
ESCOLA DE MINAS
DEPARTAMENTO DE ENGENHARIA CONTROLE E AUTOMACAO

FOLHA DE APROVAÇÃO

Wellington Resende de Araújo Júnior

Sistema de Monitoramento e Detecção de Ameaças em Redes Utilizando Aprendizado de Máquina

Monografia apresentada ao Curso de Engenharia de Controle e Automação da Universidade Federal de Ouro Preto como requisito parcial para obtenção do título de Engenheiro de Controle e Automação

Aprovada em 22 de abril de 2025

Membros da banca

[Mestre] - Gabriel Vinicios Moreira Fernandes - Orientador (Itaipu Parquetec)

[Doutor] - Agnaldo José da Rocha Reis - Coorientador (Universidade Federal de Ouro Preto)

[Graduado] - Amin Mhamad Ismail - (Itaipu Parquetec)

[Mestre] - Fernando Henrique Oliveira Duarte - (Universidade Federal de Ouro Preto)

Gabriel Vinicios Moreira Fernandes e Agnaldo José da Rocha Reis, orientadores do trabalho, aprovaram a versão final e autorizaram seu depósito na Biblioteca Digital de Trabalhos de Conclusão de Curso da UFOP em 28/04/2025.

Documento assinado eletronicamente por **Agnaldo Jose da Rocha Reis**, **PROFESSOR DE MAGISTERIO SUPERIOR**, em 28/04/2025, às 14:55, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site http://sei.ufop.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0 , informando o código verificador **0902005** e o código CRC **EE79030A**.

R. Diogo de Vasconcelos, 122, - Bairro Pilar Ouro Preto/MG, CEP 35402-163
Telefone: 3135591533 - www.ufop.br

Agradecimentos

A realização deste trabalho foi possível graças ao apoio e incentivo de diversas pessoas, às quais expresso minha sincera gratidão. Primeiramente, agradeço ao Gabriel e Agnaldo, pela orientação, paciência e dedicação ao longo deste percurso. Aos meus familiares, pelo apoio incondicional, compreensão e incentivo constante. À Mariana, por estar sempre ao meu lado, oferecendo carinho e apoio. Aos ex-alunos e moradores da República Favela, que compartilharam comigo desafios, aprendizados e momentos de motivação. À Universidade Federal de Ouro Preto e aos professores do curso de Engenharia de Controle e Automação, pelo conhecimento transmitido. Por fim, a todos que, direta ou indiretamente, contribuíram para a concretização deste trabalho, o meu mais sincero agradecimento.

Se cheguei até aqui foi porque me apoiei no ombro dos gigantes.

— Isaac Newton.

Resumo

Este trabalho tem como objetivo desenvolver, analisar e comparar métodos de classificação de dados voltados para a detecção de intrusões em redes, utilizando algoritmos de aprendizado de máquina. A crescente preocupação com a cibersegurança, impulsionada pelo aumento de ataques cibernéticos em ambientes digitais, justifica a necessidade de sistemas eficazes para defesa. Nesse contexto, busca-se desenvolver um modelo baseado em RNA e avaliar seu desempenho em relação aos modelos de *Decision Trees* e *Random Forest*. A metodologia adotada envolve o treinamento e teste dos modelos com um conjunto de dados estruturado segundo os princípios do aprendizado supervisionado, empregando métricas oriundas da matriz de confusão para avaliação dos resultados. A revisão de literatura contextualiza a evolução das técnicas de detecção de intrusões, demonstrando que, apesar do potencial das RNA, modelos mais simples podem ser mais eficientes em termos de custo-benefício. Os experimentos realizados indicam que, embora as redes neurais possam alcançar boas métricas, a escolha do modelo deve considerar não apenas a precisão, mas também fatores como complexidade e demanda computacional. Além disso, a utilização de contêineres *Docker* é destacada como uma estratégia eficaz para garantir a reprodutibilidade dos experimentos, facilitando sua replicação por outros pesquisadores. Dessa forma, este estudo contribui para o aprimoramento das estratégias de defesa cibernética, auxiliando na escolha de abordagens mais eficientes para a detecção de intrusões em redes.

Palavras-chaves: Cibersegurança, Redes Neurais Artificiais, Detecção de Intrusões, Aprendizado de Máquina, Modelos de Classificação, Análise de Ameaças, Tráfego de Rede.

Abstract

This work aims to develop, analyze and compare data classification methods aimed at detecting intrusions in networks, using machine learning algorithms. The growing concern about cybersecurity, driven by the increase in cyber attacks in digital environments, justifies the need for effective defense systems. In this context, the aim is to develop a model based on ANN and evaluate its performance in relation to model such as Decision Trees and Random Forest. The methodology adopted involves training and testing the models with a data set structured according to the principles of supervised learning, using metrics such as accuracy and confusion matrix to evaluate the results. The literature review contextualizes the evolution of intrusion detection techniques, demonstrating that, despite the potential of ANN, simpler models can be more cost-effective. The experiments carried out indicate that, although neural networks can achieve good metrics, the choice of model must consider not only accuracy, but also factors such as complexity and computational demand. In addition, the use of Docker containers is highlighted as an effective strategy for ensuring the reproducibility of experiments, facilitating their replication by other researchers. In this way, this study contributes to the improvement of cyber defense strategies, helping to choose more efficient approaches for detecting network intrusions.

Key-words: Cybersecurity, Artificial Neural Networks, Intrusion Detection, Machine Learning, Classification Models, Threat Analysis, Network Traffic.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

SIEM	*Security Information and Event Management*
IDS	*Intrusion Detection System*
SVM	*Support Vector Machine*
KNN	*K-Nearest Neighbors*
RNA	Redes Neurais Artificiais
DoS	*Denial of Service*
ML	*Machine Learning*
IA	Inteligência Artificial
PCAP	Captura de Pacote
IP	*Internet Protocol*
CSV	Valores Separados por Vírgula
TCP	Protocolo de Controle de Transmissão

Sumário

1 Introdução

Nos últimos anos, foi possível testemunhar uma acelerada transformação digital em nossa sociedade. As formas de se comunicar e armazenar informações mudaram drasticamente. Com isso, empresas e governos passaram a se preocupar com a segurança das informações que circulam no meio digital, tornando a cibersegurança uma pauta muito necessária para as nações (SVIATUN et al., 2021).

Com o advento dos computadores e da internet, além de todos os benefícios que podemos desfrutar diariamente, surgiram também inúmeras ameaças aos usuários desses novos sistemas digitais. Como alertado inicialmente por Cohen (1987), com a criação do primeiro vírus de computador e análise de seus possíveis riscos. Ele também demonstrou algumas das primeiras alternativas de defesa que viriam a dar inicio aos estudos relacionados a cibersegurança.

Nos últimos anos, houve o surgimento de diversos tipos de *softwares* maliciosos, os chamados *malwares*, com o objetivo de causar danos a sistemas, roubar informações sigilosas e obter acesso a redes e dispositivos. Os principais tipos de *malwares* incluem vírus, *worms*, *trojans* e *ransomwares*, cada um com características distintas para se propagar e gerar danos aos sistemas (SECURITY, 2024).

Os vírus são acoplados a programas ou arquivos seguros e se replicam, podendo causar danos como perda de dados ou corromper o sistema operacional por completo. Por outro lado, os *worms* são *malwares* com alta capacidade de proliferação em redes, infectando diversas máquinas. Na maioria dos casos, os *worms* são utilizados para instalar *bots* e criar um exército para realizar outros tipos de ataque, como os de Negação de Serviço (DoS). Já os *ransomwares* são ataques que visam extorquir as vítimas. Na maioria dos casos, esse tipo de ameaça criptografa todos os arquivos da máquina infectada e exige um pagamento para que seja possível recuperar os arquivos. Todas essas ameaças podem causar danos irreversíveis para pessoas e organizações, levando a enormes perdas financeiras e indisponibilidade de serviços (SECURITY, 2024).

Com o passar dos anos e a evolução dos métodos de ataque, alguns episódios chegaram a ganhar notoriedade. Como foi o caso do *Stuxnet*, um *worm* que foi propositalmente espalhado em uma região do Irã próxima às usinas de enriquecimento de urânio, com o intuito de infectar a rede interna das usinas e assim realizar um ataque em massa para comprometer a produção. O *worm* foi considerado a primeira arma de guerra cibernética (LANGNER, 2011).

Nos Estados Unidos, em 2021, o maior oleoduto do país sofreu um ataque de *ransomware* que deixou parte de seus sistemas inoperantes mediante o pagamento de uma

quantia entre 2 e 4 milhões de dólares. Ao que tudo indica os *hackers* se aproveitaram de vulnerabilidades de acesso que existiam naquele momento, já que, devido a pandemia vários engenheiros acessavam os sistemas de forma remota. Neste caso, um ataque chamado de *bruteforce* foi aplicado sob o sistema, em que são tentadas diversas combinações de usuário e senha até que se consiga tomar o acesso, como detalha uma materia publicada na BBC (BRASIL, 2021).

Segundo a Ventures (2020) estima que o mercado do cibercrime deve crescer 15% ao ano entre 2020 e 2025, chegando no último ano a quantia de 10.5 trilhões de dólares em prejuízo. Esse valor representa todos os custos diretos e indiretos causados pelos ataques, sendo considerado a maior transferência de riqueza da história. Toda essa quantia de dinheiro pode ser maior que o comércio global de drogas. Entre os cibercrimes, o mais popular é o *ransomware*. Somente este tipo de ataque teve previsão de custo de 5 bilhões de dólares em 2017, um crescimento de 15 vezes em relação a 2015, com um valor estimado de 325 milhões de dólares.

1.1 Justificativas e Relevância

Segundo KPMG (2023), as empresas que investiram em transformação digital tiveram um crescimento médio de 11% na lucratividade, entre 2021 e 2022. Com isso, podemos observar o aumento do valor associado aos sistemas digitais, atraindo a atenção dos criminosos, provocando assim, um aumento crescente das ameaças cibernéticas que impõe grandes desafios no campo de segurança da informação, uma vez que, uma ameaça cibernética pode acarretar em danos reais nos negócios, como visto no relatório Ventures (2020).

Desta forma, o tema da cibersegurança se tornou uma das principais pautas para empresas e governos, sendo até bastante discutido entre a população. Porém, devido aos altos custos necessários para manter times de tecnologia capacitados e ferramentas de ponta do mercado, muitas instituições acabam optando, ou não conseguindo, manter um alto padrão de defesa contra esses tipos de ataques.

Com isso, é de suma importância o desenvolvimento de ferramentas de análise de ameaças, para auxiliar times de tecnologia na detecção e combate de ameaças de forma precoce, minimizando ao máximo os danos, que muitas vezes podem ser irreparáveis.

1.2 Objetivos

Objetiva-se desenvolver, analisar e comparar métodos de classificação de dados para diagnóstico de intrusão de redes, sendo o foco do trabalho o desenvolvimento de uma rede neural artificial para detecção desses ataques, a fim de avaliar o seu desempenho

diante de métodos de classificação convencionais. Serão analisados 3 tipos de ataque de rede diferentes, sendo eles: DoS, *PortScan* e *Bruteforce*.

Com base no objetivo geral apresentado, serão definidos os seguintes objetivos específicos:

1. Implementar um ambiente em *docker* para simular uma aplicação de software e sua interação com componentes, tráfego benigno e maligno.
2. Desenvolver e treinar dois modelos de classificação utilizando as técnicas de *Decision Trees* e *Random Forest*, coletando métricas de desempenho para análise futuras.
3. Criar e treinar um modelo de rede neural artificial do tipo *Perceptron* Multicamadas, coletando suas métricas para análises.
4. Obter a estrutura base para o desenvolvimento de uma plataforma de monitoramento de redes em tempo real de baixo custo para uso em pequenos empreendimentos.

1.3 Materiais e Métodos

Objetiva-se com este trabalho analisar a eficácia de modelos de aprendizado de máquina para detecção de intrusão de redes, ademais, analisar a diferença de desempenho entre modelos de redes neurais artificiais, *Decision Tree* e *Random Forest*. Desta forma teremos uma abordagem quantitativa dos métodos utilizados, visto que será baseada em dados experimentais e conta com análises estatísticas feita por ferramentas adequadas.

Para treinamento e teste dos modelos, será utilizado uma base de dados própria, adquirida através da Captura de Pacotes de Rede (PCAP) em ambiente *docker* simulando uma aplicação *backend* em *Python Flask*, a qual receberá tráfego de requisições benignas e malignas, além de se comunicar com um banco de dados *postgres* e um *cache redis*. O ambiente proposto simulará máquinas com *Linux Ubuntu*. Para selecionar as propriedades do tráfego que seriam armazenadas foi utilizando como referência o trabalho de Sharafaldin, Lashkari, Ghorbani et al. (2018).

Todos os códigos serão criados através da linguagem *Python* e seus *frameworks*. Os modelos de aprendizado de máquina, *Decision Tree* e *Random Forest* serão criados utilizando a biblioteca *sklearn*, que possui a implementação de diversos tipos de modelos, tratativas de dados e métricas. Já as redes neurais serão desenvolvidas por meio do pacote *tensorflow*, especializado em modelos deste tipo. Por fim, o sistema de coleta e monitoramento em tempo real será desenvolvido com base na ferramenta *pyshark*, que fará a coleta do tráfego de rede na máquina.

Por fim, para análise dos resultados de cada um dos modelos criados serão utilizadas algumas métricas de desempenho de modelos de classificação, sendo elas: Acurácia e a

matriz de confusão, extraindo dessa última as métricas de especifidade, precisão e o *F-Score*.

1.4 Organização e estrutura

O Capítulo 1 aborda um breve histórico de ataques em ambientes industriais. Além disso, também é enfatizado o tamanho do mercado de cibercrimes e sua constante evolução. O Capítulo 2 discorre sobre trabalhos similares, trazendo diversas abordagens do problema utilizadas em estudos anteriores, além de considerar também seus principais benefícios e dificuldades. O Capítulo 3 apresenta detalhes sobre o desenvolvimento dos modelos e conjuntos de dados utilizados para treinamento e testes. Em seguida, no Capítulo 4 são passados os resultados obtidos por todos os modelos produzidos através da metodologia apresentada. O Capítulo 5 fala sobre o comparativo entre os métodos abordados e sobre a continuidade do trabalho em estudos futuros e as conclusões deste trabalho.

2 Revisão de literatura

2.1 Trabalhos Relacionados

Cinque, Cotroneo e Pecchia (2018) abordam o problema de detecção de ameaças na rede através da construção de traços de comportamento para criar bases de comportamento regular do sistema, e utilizam de Alocação Latente de Dirichlet (LDA) para criar modelos estatísticos generativos que permitem encontrar relações entre dados observados, mencionando também o uso de inteligência sobre os dados para detecção de ameaças. Além disso, cita sobre a dificuldade de trabalhar com registros de texto não estruturado para identificação de ataques em um sistema crítico e a participação importante de recursos cognitivos humanos na proteção de computadores.

Moukafih, Orhanou e El Hajji (2020) apresenta uma solução de detecção de invasões em sistemas SIEM majoritários baseados em confiabilidade, o modelo proposto combina diversas redes neurais *feedfoward* simples como *weak learners*, proporcionando boa capacidade de acerto utilizando poucos recursos computacionais. A pesquisa também aborda o desafio em minimizar a necessidade de recursos de processamento exigidos por sistemas desse tipo, além da exigência de técnicas de pré-processamento adequadas para cada contexto. Ao fim o modelo se provou bastante eficaz e com alta precisão, sendo uma solução promissora para detecção de invasões.

Ch et al. (2020) apresenta um sistema que utiliza técnicas de aprendizado de máquina como Naive Bayes e KNN, sendo o primeiro para classificação e o segundo para agrupamento, a fim de identificar e classificar os ataques cibernéticos. A pesquisa também trás a necessidade de generalizar os algoritmos de detecção baseando-se na coleta de recursos e, por fim, ressalta o potencial do aprendizado de máquina para essa tarefa.

Entre os modelos apresentados no trabalho de Ch et al. (2020), o que mais se destaca em relação a abordagem do presente trabalho seria o modelo de *Naive Bayes*, que realiza classificação dos diversos tipos de ataques. O modelo apresentou ótimos resultados entre os 3 tipos de ataques analisados (1).

Ch et al. (2020) também trouxe um detalhamento de métricas similar ao utilizado no presente trabalho, onde foi abordado *F1-Score*, precisão e *recall* (2).

Além do modelo de classificação principal também foram desenvolvidos outros 4 modelos a efeito de comparação, entre eles o que apresentou melhor acurácia foi a Regressão Logística com 99,38%, em contra partida o modelo *Random Forest* apresentou 80,69%, sendo o pior método, como mostra a figura 3.

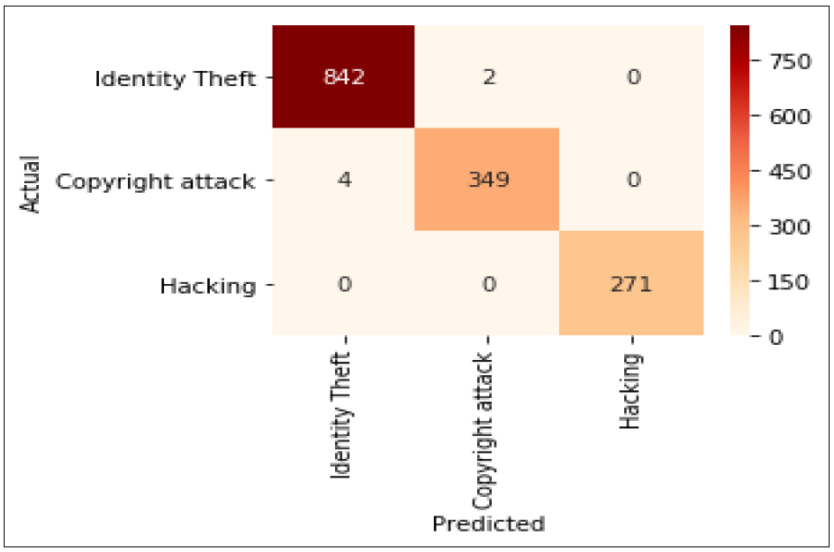

Figura 1 – Matriz de confusão do modelo *Naive Bayes* de acordo com o tipo de ataque. Fonte: Ch et al. (2020)

	precision	recall	f1-score	support
Identity Theft	1.00	0.99	0.99	241
Copyright attack	0.97	1.00	0.98	95
Hacking	1.00	1.00	1.00	84
avg / total	0.99	0.99	0.99	420

Figura 2 – Tabela de métricas do modelo de *Naive Bayes* de acordo com o tipo de ataque. Fonte: Ch et al. (2020)

```
model_name
LinearSVC                   0.992371
LogisticRegression          0.993803
MultinomialNB               0.989516
RandomForestClassifier      0.806923
Name: accuracy, dtype: float64
```

Figura 3 – Taxa de acurácia para cada modelo. Fonte: Ch et al. (2020)

Pelletier e Abualkibash (2020) faz uso da base de dados CICIDS 2017, e sugere algumas tratativas de dados para melhorar a eficiência dos modelos de aprendizado, como seleção das melhores variáveis para reduzir a quantidade de argumentos, tratativas para dados faltantes e, por fim, apresenta o desempenho de alguns tipos de modelos para predição dos dados do conjunto.

No trabalho de Pelletier e Abualkibash (2020) foi treinado um modelo de RNA com 500 épocas, o qual obteve uma acurácia de 96,53%, utilizando como fonte de dados a base CICIDS-2017, já mencionada. O mesmo modelo, treinado com 50 épocas apresentou acurácia de 87,79%. O estudo também trouxe uma visão detalhada do resultado do modelo de acordo com o tipo de ataque, onde foi possível notar uma maior dificuldade em detectar corretamente ataques do tipo *DoS Hulk*, *SQL Injection* e *Heartbleed*, em ordem, como mostra a figura 4.

TABLE IV. Attack Detection Metrics for 500 Iteration Neural Network

Attack	Attack Detection Metrics For 500 iteration Neural Network		
	Number of Connections Correctly Predicted	Total Connections	% Correct
Benign (Normal Traffic)	1839789	1841023	99.933%
DDoS	128025	128027	99.998%
Portscan	158898	158930	99.980%
Bot	1960	1966	99.695%
Infiltration	34	36	94.444%
Web Attack: Brute Force	1505	1507	99.867%
Web Attack: XXS	652	652	100%
Web Attack: SQL Injection	19	21	90.476%
DoS Slowloris	5582	5796	96.308%
DoS Slowhttptest	5381	5499	97.854%
DoS Hulk	200067	231073	86.582%
DoS GoldenEye	10175	10293	98.854%
Heartbleed	10	11	90.909%

Figura 4 – Tabela de resultados do modelo de RNA de acordo com o tipo de ataque. Fonte: Pelletier e Abualkibash (2020)

Por fim, os autores realizam um comparativo com o método de *Random Forest*, que apresentou acurácia geral de 96,24%, e se percebeu um desempenho mais consistente deste tipo de modelo para os diferentes ataques presentes na base, como mostra a figura 5, que detalha os resultados de cada um.

Em Muhammad, Sukarno e Wardana (2023) foi proposto a integração de um SIEM e um sistema IDS que utilizavam aprendizado de máquina através de *Support Vector Machine* para detecção de ataques baseada em análises em tempo real dos dados de rede. O sistema foi construído através de ferramentas *open-source* e preparado para aplicações industriais. O mesmo possui monitoramento em tempo real e envio de alertas para os times responsáveis quando alguma anomalia é detectada.

2.2 Fundamentação Teórica

Nesta seção são apresentados conceitos teóricos necessários para entender o funcionamento do modelo de rede neural construído neste trabalho. Na subseção 2.2.3 é introduzido o conceito de RNA e seu funcionamento de forma geral.

TABLE V. Attack Detection Metrics for randomForest Training

Attack	Attack Detection Metrics For randomForest Training		
	Number of Connections Correctly Predicted	Total Connections	% Correct
Benign (Normal Traffic)	1834784	1841023	99.661%
DDoS	127844	128027	99.857%
Portscan	158798	158930	99.917%
Bot	1912	1966	97.253%
Infiltration	33	36	91.167%
Web Attack: Brute Force	1477	1507	98.009%
Web Attack: XXS	649	652	99.540%
Web Attack: SQL Injection	20	21	95.238%
DoS Slowloris	5640	5796	97.309%
DoS Slowhttptest	5486	5499	99.764%
DoS Hulk	217813	231073	94.262%
DoS GoldenEye	10018	10293	97.328%
Heartbleed	9	11	81.818%

Figura 5 – Tabela de resultados do modelo de *Random Forest* de acordo com o tipo de ataque. Fonte: Pelletier e Abualkibash (2020)

2.2.1 *Decision Trees*

As *Decision Trees* são modelos de classificação construídos a partir de nós e arcos (FÜRNKRANZ; GAMBERGER; LAVRAČ, 2012). Os nós podem ser internos ou externos, tendo representações diferentes de acordo com o tipo. Cada nó interno representa um teste de uma característica dos dados analisados (ex: tamanho do animal < 1 metro), e os nós externos representam os rótulos presentes no conjunto de dados (ex: Gato). Por fim, os arcos representam os resultados dos testes realizados nos nós internos (ex: verdadeiro/falso). A árvore é percorrida de cima para baixo até alcançar algum dos nós externos, também conhecido como nó-folha.

Na figura 6, é possível ver uma Árvore de Decisão com 3 nós internos, que analisam condições de *Marital Status*, *Sex* e *Has Children*, 4 arcos e 5 nós externos, que representam os rótulos *Yes* e *No*.

Os modelos de *Decision Tree* são muito utilizados para mineração de dados, isso se deve a capacidade de trabalhar com dados de diversos tipos, sejam valores numéricos ou rotulados, e por se tratar de um modelo de fácil entendimento da sua representação (estrutura dos nós e arcos). Além disso, os modelos de árvore são mais rápidos de se treinar se comparados aos de redes neurais artificias, também abordados neste trabalho.

2.2.1.1 Implementação e Criação do Modelo

Para se determinar os testes utilizados, e deste forma construir a árvore, o algoritmo mais utilizado é o chamado *divide and conquer*, que realiza a divisão do todo em subconjuntos de maneira recursiva. Esta abordagem se inicia definindo a árvore de decisão mais geral possível, com apenas um nó inicial. A partir deste nó, o algoritmo refina a

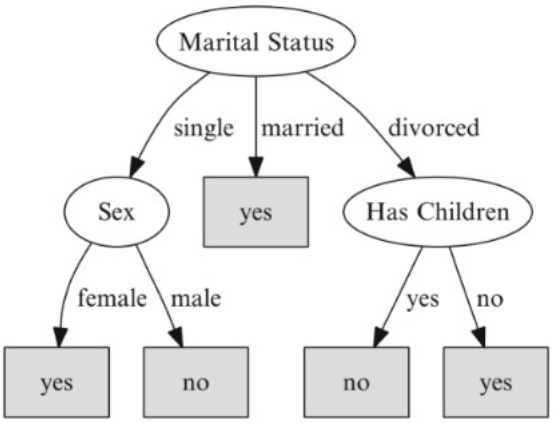

Figura 6 – Exemplo de um modelo *Decision Tree*. Fonte: Fürnkranz, Gamberger e Lavrač (2012)

árvore até que cada subconjunto pertença a um único rótulo.

O fator mais importante ao se criar este tipo de modelo é a escolha dos testes de cada nó. Uma condição é definida de forma que se tenha a maior quantidade possível de instâncias dos dados pertença a um único rótulo. Um dos critérios mais utilizados para a seleção da característica que se tornará um nó da árvore é o cálculo do Ganho de Informação.

Este tipo de modelo é muito suscetível ao problema de *overfitting*, devido a isso é necessário, após a construção da árvore, realizar a 'poda' da mesma. Algumas circunstâncias nos dados utilizados no treinamento podem levar a criação de partes da árvore que prejudicam a capacidade de generalização.

A 'poda' pode ser realizada de algumas formas, as mais comuns são a pré e a pós-poda. Na primeira a construção da árvore é interrompida por um critério de parada pré-definido, logo, a árvore é limitada durante a construção. Na segunda, após finalizada, a árvore tem algumas sub-árvores substituídas por um único nó externo, geralmente definido pela classe mais comum na sub-árvore removida.

2.2.2 *Random Forest*

As *Decision Trees* são uma ótima opção de classificadores devido a sua alta velocidade de execução em relação a outros modelos, porém, estes algoritmos podem apresentar problemas em relação a complexidade, levando a baixa capacidade de generalização para dados não vistos e até mesmo do conjunto de treinamento. Com isso em mente, Ho (1995) propôs o modelo de *Random Decision Forest*, baseando-se nos princípios da modelagem estocástica, o método busca construir um classificador baseado em árvores que consiga melhores resultados para dados não conhecidos e de treinamento.

O método constrói múltiplas árvores em subespaços selecionados aleatoriamente, estas árvores em diferentes subespaços generalizam sua classificação de maneiras complementares,e ao fim, combinam seus resultados de forma a torná-lo mais assertivo (HO, 1995). O resultado final do modelo, é basicamente o mais escolhido entre às árvores que o constituem, assim como mostra a figura 7

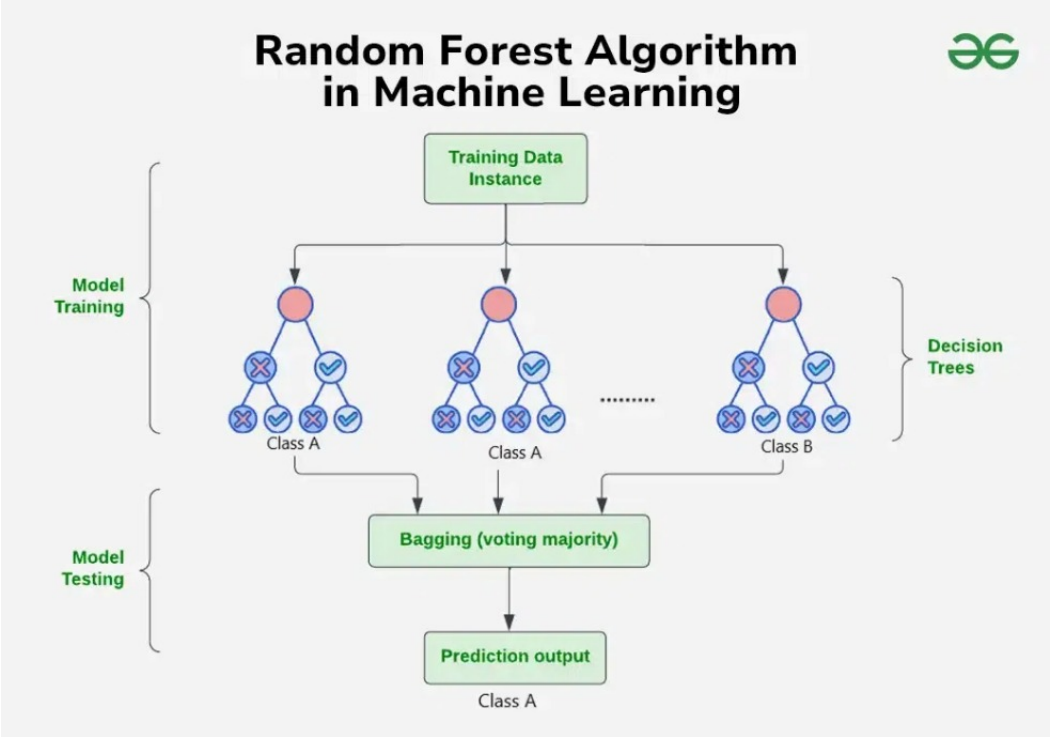

Figura 7 – Exemplo de um modelo *Random Forest*. Fonte: Geeks (2025)

2.2.3 Redes Neurais Artificiais

Normalmente problemas de programação seguem regras lógicas para fornecer seus resultados. Entretanto, quando o problema a ser resolvido com código não é necessariamente lógico, os algoritmos tradicionais acabam se tornando inviáveis. Para estes casos devemos utilizar algoritmos que consigam simular o procedimento intuitivo dos seres humanos de resposta probabilísticas.

Com isso em mente,um dos algoritmos desenvolvidos são os de Redes Neurais Artificiais(RNA), que buscam simular o comportamento dos neurônios humanos, através de uma arquitetura que acaba por imitar o cérebro, com camadas de neurônios artificiais que recebem e transmitem informação de forma semelhante a uma estrutura de neurônios naturais. Esse tipo de algoritmo não possui um procedimento definido de forma explícita a ser seguido.Para desenvolver esta capacidade ”cognitiva”as redes neurais utilizam conjuntos de treinamento que, através dos algoritmos de aprendizado implementados, serão capazes

de solucionar novas instâncias do problema. Essa característica é chamada de generalização (BRAGA; LUDERMIR; CARVALHO, 2000).

Desta forma, segundo Braga, Ludermir e Carvalho (2000), redes neurais artificiais são sistemas distribuídos com diversas unidades de processamento simples, chamadas nodos ou nós, e que possuem uma função matemática implementada, chamada função de ativação. Estes nós estão organizados em camadas, sendo a primeira e a última chamadas de camada de entrada e saída respectivamente, e as demais, chamadas de camadas ocultas. Por fim, esses nós são interligados por conexões, onde cada uma possuí um peso atribuído.

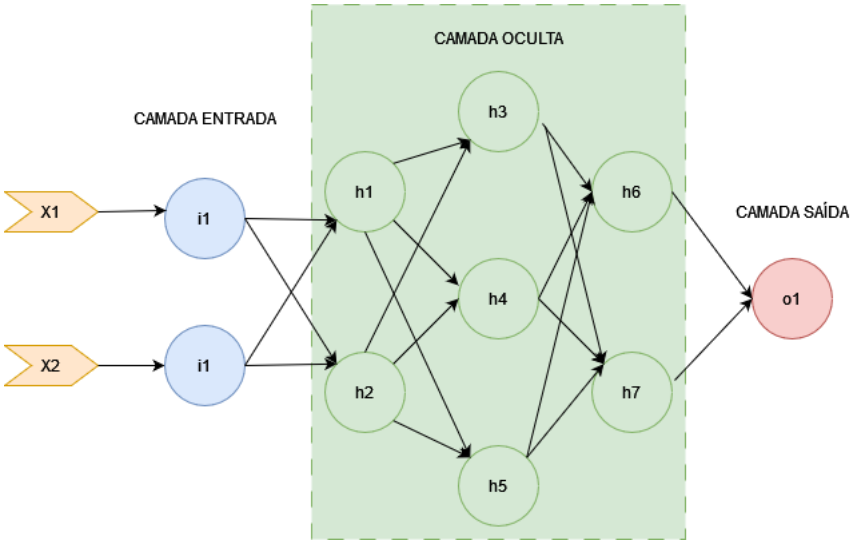

Figura 8 – Modelo de RNA contendo as 3 camadas descritas e as conexões para interligação.

2.2.3.1 Camadas de uma rede neural e os tipos de modelos de RNA

A camada de entrada é responsável por receber os dados do mundo externo, desta forma, seus nós transformam os dados para atender as limitações da rede, analisam e encaminham para as próximas camadas. Em seguida, as camadas ocultas recedem os dados da camada anterior e fazem um novo processamento neles. Estas camadas podem receber dados da camada de entrada ou de outras camadas ocultas. Por fim, a camada de saída fornece o resultado final dos dados processados, ela pode conter um ou mais nós, a depender do tipo de problema abordado. Segundo o guia de desenvolvedores da AWS (AMAZON..., s.d.), existem 3 tipos de modelos em algoritmos de ML: O de classificação binária, onde só existem 2 saídas possíveis, o modelo de classificação multiclasse, nos quais existem diversas saídas predefinidas possíveis, como é o caso do modelo desenvolvido neste trabalho. Por último, o modelo de regressão, onde existem infinitas saídas possíveis de um valor numérico. Cada um destes modelos trará uma configuração para a camada de saída diferente, a depender da quantidade de saídas previstas.

2.2.3.2 Funções de ativação

Para tornar possível o aprendizado de relações não-lineares entre as variáveis, é de fundamental o uso das funções de ativação, como mostra Hagan, Demuth e Beale (1997). Estas funções são componentes matemáticos que guiam o funcionamento dos neurônios artificias e tratam para que os dados de saída estejam condizente com um limite de amplitude permitido para o problema proposto. Além disso, as funções de ativação são determinantes para as capacidades de aprendizado, velocidade de convergência e o desempenho geral do modelo, sendo crucial a escolha de uma função apropriada para a arquitetura do modelo de RNA.

Para Hagan, Demuth e Beale (1997), entre os principais tipos de função de ativação estão: A *Sigmoid*, onde os valores de entrada são comprimidos entre 0 e 1, fornecendo gradientes suaves, porém sofrem problemas quando implementadas em redes profundas. A função Tangente Hiperbólica, que por sua vez mapeia os valores de entrada para o intervalo [-1, 1], centrada em zero, o que pode ajudar na convergência mais rápida de modelos, entretanto, esse tipo de função também enfrenta problemas em redes profundas. A ReLU (Unidade Linear Retificada), define quaisquer valores negativos como zero e mantém os positivos inalterados (linearidade), é uma função computacionalmente eficiente e ajuda a resolver o problema que as duas anteriores possuem em redes profundas, porém, em alguns casos pode ocorrer a ”morte”dos neurônios, tornando-os inativos e interrompendo o aprendizado. Para evitar esse efeito de ”morte”da função ReLU, foi desenvolvida a *Leaky* ReLu, permitindo um pequeno gradiente para valores negativos e evitando que um neurônio se torne completamente inativo, além disso, pode levar a uma convergência mais rápida do modelo.

2.2.3.3 Peso dos nós e o algoritmos de otimização

Os pesos por sua vez, fazem a representação numérica da conexão entre neurônios, variando entre -1 e 1. Eles funcionam como uma forma de armazenar o conhecimento adquirido e são atualizados por meio de algoritmos iterativos buscando atingir um ponto ótimo. O principal algoritmo utilizado para esse fim, é o de Retropropagação do erro, ou *Backpropagation*, como é mais conhecido.

Como descrito por Hagan, Demuth e Beale (1997), o algoritmo de *Backpropagation* possui três etapas, na primeira, que pode ser chamada de *Forward Pass*, os dados de entrada são inseridos na rede, que foi criada com pesos aleatórios, e calcula a saída do modelo. Cada neurônio deverá calcular sua saída através da soma ponderada de suas entradas e aplicar a função de ativação para produzir a saída.

O próximo passo é calcular o erro entre a saída prevista e o valor real. Para isso podemos utilizar alguma métrica, como por exemplo o Erro Quadrático Médio, muito

utilizado em problemas de regressão, ou a Entropia Cruzada, para modelos de classificação.

Por fim, ocorre a etapa de retropropagação, essa etapa tem como objetivo propagar o erro para as camadas de trás da rede para atualizar os pesos e minimizar o erro. Neste processo temos o cálculo de gradientes da função de perda selecionada em relação aos pesos e por fim, o ajuste dos pesos da rede na direção oposta ao gradiente, buscando minimizar o erro.

2.2.3.4 Treinamento de modelos de RNA

Existem dois principais métodos para o treinamento de modelos, supervisionado e não-supervisionado.Como descrito por Hagan, Demuth e Beale (1997), eles se distinguem pela configuração do conjunto de dados utilizado para o treinamento. No primeiro é necessário um conjunto de dados que pode ser separado em duas partes, as variáveis independentes (entradas) e os rótulos, os quais podem ser comparados com os valores previstos pelo modelo e apontar se estamos na direção certa. Porém, nos treinamentos não-supervisionados não possuímos um conjunto de rótulos, apenas os dados de entrada do modelo, fazendo com que o modelo aprenda padrões nos dados de entrada sem um direcionamento explicito. Entretanto essas abordagens possuem suas limitações, como trata Hagan, Demuth e Beale (1997), o aprendizado supervisionado podem ser utilizados para treinar modelos de qualquer tipo, tanto de classificação quanto regressão, já o método não-supervisionado, o foco é em reconhecer padrões entre conjuntos de dados, sendo utilizado, principalmente, para clusterização.

Para se obter um modelo de rede neural com aprendizado supervisionado, o qual é abordado neste trabalho, devemos dividir o conjunto de dados em duas partes: dados de treinamento e dados de teste. O primeiro será utilizado para treinar o modelo e calcular os pesos e o segundo para avaliar o desempenho final de generalização da rede desenvolvida, comparando os resultados obtidos pela predição da RNA com as saídas reais, como abordado por Lek e Guégan (1999). Em alguns casos, muito comum para desenvolvimento de RNAs, é utilizado um terceiro conjunto de dados, chamado dados de validação, que é utilizado durante o treinamento da rede para avaliar seu desempenho previamente, sendo importante para a boa definição de parâmetros da RNA criada (HAGAN; DEMUTH; BEALE, 1997).

Lek e Guégan (1999) também ressalta a importância da qualidade do conjunto de dados utilizados no treinamento de uma RNA. Além de possuir um conjunto de dados volumoso, é fundamental que eles sejam abrangentes, contendo o máximo de possibilidades de resultados. Desta forma teremos um modelo bem treinado e capaz de generalizar situações reais.

2.2.3.5 Critérios de parada

No aprendizado de máquina é necessário definir critérios de parada adequados no desenvolvimento da RNA, essa escolha previne o *overfitting* dos modelos, otimiza o tempo de treinamento e busca garantir a convergência do modelo para melhor solução possível. Algumas das condições de parada mais utilizadas para RNA são o número máximo de épocas e a validação de perdas. Na primeira, é definida um número máximo de épocas para evolução da rede, ao atingir o número definido o treinamento é interrompido. Já no caso das perdas, a evolução é finalizada quando se percebe um aumento nas perdas do modelo ao longo de várias épocas, esse fator indica que o modelo está superajustado (*overfitting*) aos dados de treinamento, impactando na capacidade de generalização.

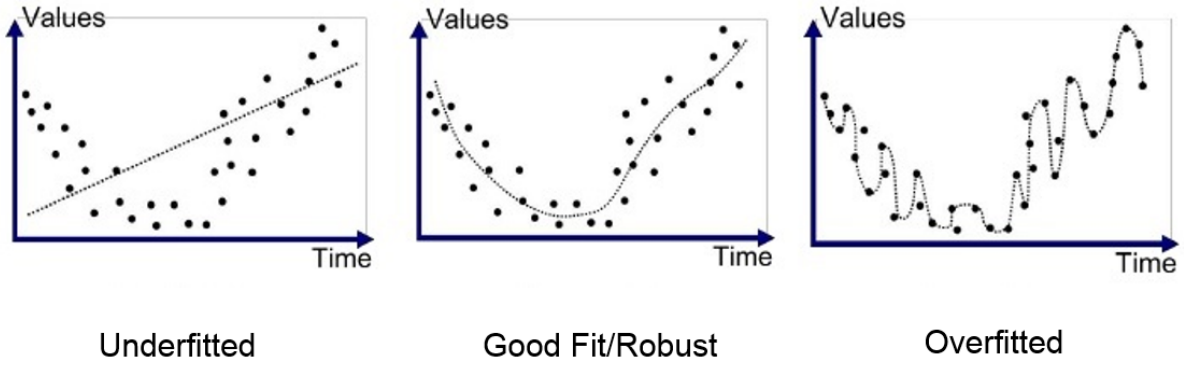

Figura 9 – Exemplo de curvas com overfitting e underfitting. Fonte: Branco (2020)

O *overfitting* e o *underfitting* são condições que devem ser evitadas no treinamento de qualquer modelo de *machine learning* pois criam graves problemas de desempenho na generalização. O primeiro se trata de um caso de superajuste aos dados de treinamento, na qual o modelo não tem capacidade de tratar novos dados. Já no *underfitting* o modelo não consegue sequer predizer os dados de treinamento, ficando subajustado, pode acontecer ao tentar impor um tipo de modelo, por exemplo, regressão linear, para um problema que possuí outra característica, como mostra a figura 9 (BRANCO, 2020).

3 Desenvolvimento

O trabalho proposto se dividiu em cinco etapas. A primeira consistiu na escolha do conjunto de dados e suas respectivas análises e pré-processamento. As três etapas seguintes consistiram na obtenção dos modelos de aprendizado de máquina proposto, e finalizando com a exportação destes modelos para uso na última etapa. Por fim, foi desenvolvido um sistema de monitoramento do tráfego de rede e detecção de ataques a partir da coleta de captura de pacotes de rede. Todo o processo é ilustrado na figura 10 onde são trazidos as etapas e as ferramentas nas quais foram desenvolvidas.

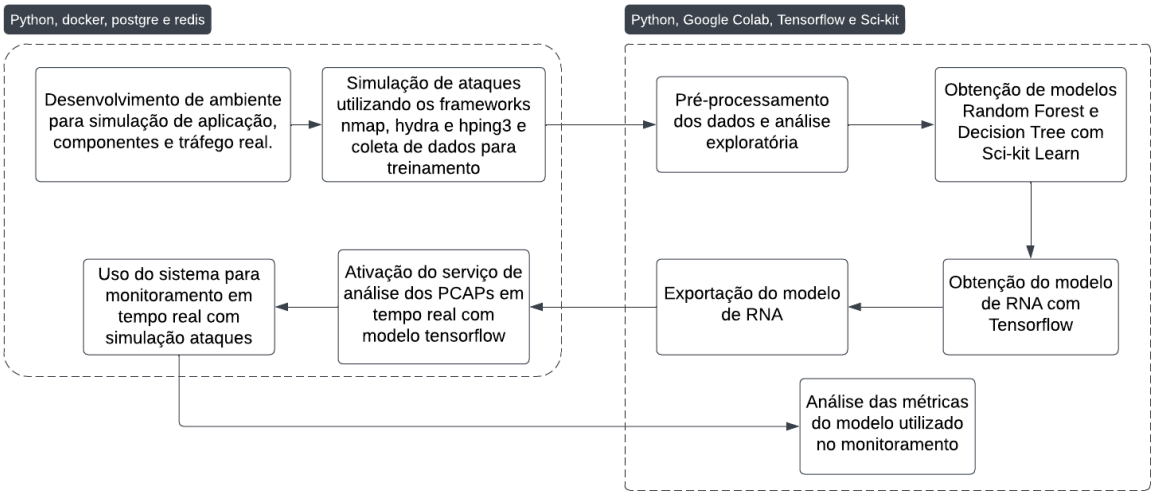

Figura 10 – Fluxograma do processo de desenvolvimento. Fonte: Autor.

3.1 Uso do conjunto de dados CICIDS2017

Inicialmente foi proposto o uso do conjunto de dados CICIDS2017 para treinamento dos modelos utilizados no trabalho. Este *dataset* foi desenvolvido por Sharafaldin, Lashkari, Ghorbani et al. (2018) e compõe uma série de estudos relacionados a cibersegurança da Universidade de *New Brunswick*. Os dados que compõe o conjunto foram obtido a partir da captura de tráfego de rede de diversos computadores durante 5 dias. Foram capturados e classificados tráfego benign e mais 14 tipos de ataques, sendo eles: *Brute Force FTP*, *Brute Force SSH*, *DoS*, *Heartbleed*, *Web Attack*, *Infiltration*, *Botnet* e *DDoS*, resultando em 2,83 milhões de pacotes capturados com 80 parâmetros registrados.

Através desse modelo foram treinados 2 modelos de redes neurais artificiais do tipo *Perceptron* Multicamadas, a primeira contendo 15 rótulos para classificação (benigno e 14 tipos de ataques), e a segunda onde o conjunto foi tratado com classificação binária

(benigno e maligno), sem distinção entre os ataques. Ao treinar o modelo foi realizada a seleção dos 10 principais parâmetros para classificação segundo Pelletier e Abualkibash (2020), sendo eles por ordem de relevância:

1. *Init Window Bytes Forward*: Tamanho médio das janelas de pacotes iniciais enviados;
2. *ACK Flag Count*: *Flag* ACK presente no pacote;
3. *Forward Packets per second*: Quantidade de pacotes enviados por segundo;
4. *Flow Packets per second*: Quantidade de pacotes entre dois pontos por segundo, independente de sentido;
5. *Flow IAT Max*: A maior diferença de tempo entre pacotes trocados por dois pontos;
6. *Flow IAT Min*: A menor diferença de tempo entre pacotes trocados por dois pontos;
7. *Flow duration*: A difença de tempo entre o primeiro e o último pacote trocado entre dois pontos;
8. *Init Window Bytes Backward*: Tamanho médio das janelas de pacotes iniciais recebidos;
9. *Subflow Backward Bytes*: Quantidade total de *bytes* recebidos;
10. *Flow IAT Mean*: A diferença de tempo média entre pacotes trocados por dois pontos;

Os modelos apresentaram resultados significativos nas duas versões, com acurácias superiores a 90%. Entretanto, ao tentar reproduzir o processo de coleta das informações em ambiente controle foi encontrado grande dificuldade, devido a falta de detalhamento do procedimento para obtenção de cada uma das variáveis, ficando aberto a interpretações equivocadas.

Em decorrência disso, optou-se por seguir uma nova abordagem e realizar a coleta de dados próprios, utilizando ambiente de simulação em *docker*. Além disso, optou-se por reduzir o escopo a somente três tipos de ataques, *portscan*, *DoS* e *Bruteforce*, por se tratarem de estratégias de fácil replicação e comumente utilizados.

3.2 Aquisição e pré-processamento de dados

3.2.1 Ambiente para simulação de aplicação *Python Flask*

Para que seja possível simular uma aplicação sendo atacada em condições similares a realidade, foi desenvolvido um ambiente utilizando *docker* que deve simular uma aplicação *Python Flask* com 3 rotas, um sistema de monitoramento de tráfego utilizado *PyShark*, um

banco de dados *PostgreSQL*, um *cache* e uma fila *Redis*, um serviço de processamento dos dados e dois serviços para simulação de tráfego destinado a aplicação, sendo um maligno e outro benigno. Todos os serviços foram criados a partir de um único *script docker-compose*, resultando em 8 serviços diferentes dentro do contêiner *docker*.

Para simular um ambiente próximo da realidade, foram desenvolvidas duas redes do tipo *bridge* no contêiner, de forma a isolar os "usuários"da aplicação em um rede externa e os componentes do sistema em uma rede interna. As redes foram denominadas como *frontend* e *backend*, respectivamente.

O principal serviço criado acomoda a aplicação *Flask* e o sistema de monitoramento de rede, capturando todo o tráfego de entrada e saída da máquina. Nele foi utilizado a imagem "*python:3.9-slim*"e instalados alguns pacotes da linguagem *python* conforme necessidade dos serviços. Este serviço também ficará atrelado às duas redes internas do docker, *backend* e *frontend* e deixará a porta 8000 disponível para conexões externas, para que possa receber requisições de outras máquinas e se comunicar com os serviços dos demais componentes. Por fim, esse serviço será limitado em duas unidades de processador (CPU) e 1GB de memória RAM.

Foram necessários dois serviços *Redis* diferentes, porém, suas criações foram similares. Ambos utilizam a imagem "*bitnami/redis:latest*", criam um volume próprio para salvar os dados em memória caso o contêiner seja encerrado e possuem limitações de consumo de hardware em 1 unidade de processamento e 300MB de memória RAM. O primeiro serviço é utilizado como cache da aplicação *Flask* e armazena o valor de um somatório e só possui acesso a rede *frontend*, já o outro, é utilizado para criação de filas de processamento para as diversas partes do sistema de monitoramento em tempo real e se comunica apenas com as máquinas da rede *backend*.

O último serviço relacionado a armazenamento de dados acomoda um banco de dados relacional *PostgreSQL* e serve como registro dos eventos de rede do sistema de monitoramento, armazenando os dados extráidos dos pacotes de captura e também os dados obtidos após processamento para uso no modelo de rede neural de classificação do tipo de tráfego, utilizando a imagem *docker "bitnami/postgresql:latest"*. Este serviço acessa somente a rede *backend* e espelha sua porta 5000 na 5432 da máquina *host (Windows)*, para que seja possível acessar o banco no *pgAdmin 4* e monitorar o processo. Esta máquina pode consumir até uma unidade de processamento e 1,5GB de memória RAM.

Para que o processamento dos dados obtidos não concorresse na mesma máquina que a captura e a aplicação rodam, foram criados dois outros serviços capazes de se replicar de acordo com a demanda e capacidade da máquina *host* e acelerar o processamento das informações. Ambos só possuem acesso a rede *backend* e não abrem nenhuma de suas portas. O serviço denominado *Process* é responsável por obter os dados utilizados no modelo através de consultas no banco de dados envolvendo as conexões entre duas máquinas

distintas, por executar consultas simples esse serviço e cada uma de suas réplicas pode consumir até uma unidade de CPU e 300MB de memória RAM. O *Analyze* é responsável por receber esses dados e tratar para serem utilizados no modelo de classificação *Tensorflow*, e está limitado ao consumo de hardware de uma unidade de processamento e 1GB de RAM.

As requisições benignas direcionadas a aplicação *Flask* são originados em um serviço que utiliza a imagem "*python:3.9-slim*"e executa um *script Python* que faz requisições na rota "/sum"com um intervalo de tempo aleatório. Este possui acesso a rede *frontend* e não compartilha nenhuma de suas portas. Quanto ao hardware, ele está limitado a consumir uma unidade de CPU e 100MB de memória RAM.

Por fim, o último serviço a ser descrito é o responsável por todos os ataques destinado a aplicação, nele foi utilizado uma imagem limpa do *Linux Ubuntu* e todas as ferramentas necessárias foram instaladas via script *Dockerfile*, como: *metasploit*, *hydra*, *hping3* e *nmap*. Além de outros componentes auxiliares, como: *curl*, *nano*, *wget* e *git*. Por exigir um certo nível de processamento, essa máquina tem permissão para consumir 3 unidades de CPU e 1,5GB de memória RAM.

O contêiner que carrega todos esses serviços foi executado em uma máquina com sistema operacional *Windows 11 Pro*, processador *12th Gen Intel(R) Core(TM) i7-1255U 1.70 GHz* com 10 núcleos e 12 processadores lógicos e 16GB de memória RAM.

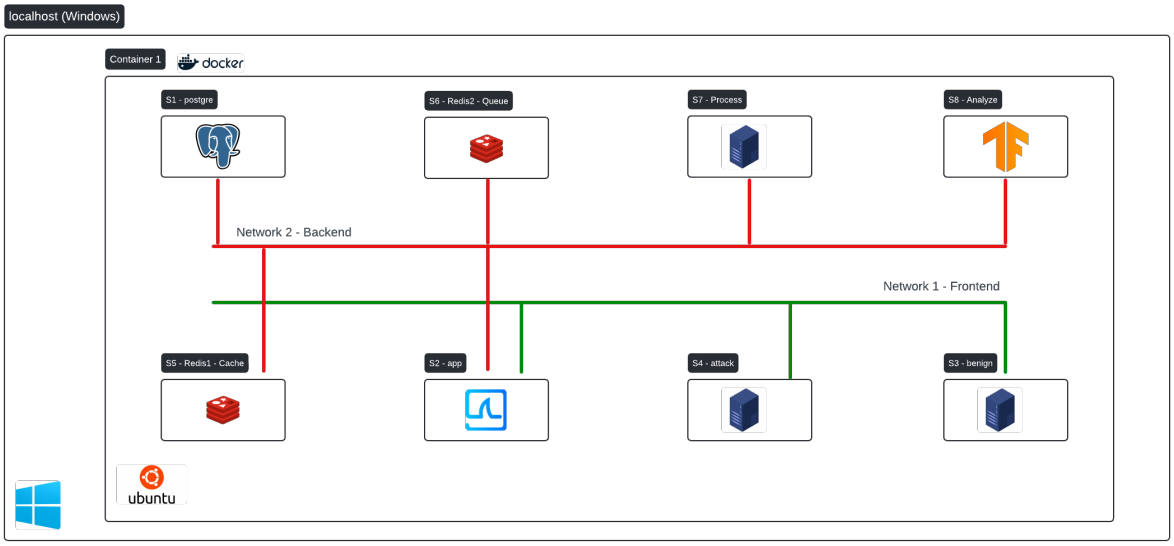

Figura 11 – Arquitetura do Contêiner *Docker*. Fonte: Autor

3.2.2 Aplicação *Flask* e monitoramento *Pyshark*

3.2.2.1 Aplicação *Flask*

A aplicação *Flask* utilizada possui três possíveis rotas para atender aos diferentes tipos de ataque utilizado, sendo elas: */hello*, */sum* e */login*. A primeira não executa nenhuma ação específica, apenas retorna uma mensagem de *"Hello World"*. A segunda por sua fez, deve receber no corpo um número qualquer e buscar pelo atual valor do *cache Redis*, somar os dois valores e atualizar o valor do *cache* para o novo resultado, feito isso deve retornar para o solicitante o novo somatório. A última rota foi desenvolvida para atender as necessidade de simulação de um ataque do tipo *Bruteforce* e recebe dados de usuário e senha, compara com os valores definidos como corretos e retorna uma mensagem de erro ou sucesso na tentativa de acesso.

3.2.2.2 Monitoramento *Pyshark*

Para coletar o tráfego de rede na máquina da aplicação *Flask*, foi desenvolvido um programa que utiliza o pacote *Pyshark*, baseado na ferramenta *Tshark* e integrado à linguagem *Python*. Inicialmente, são definidos, através do *hostname* das máquinas, quais IPs devem ser ignorados da análise, como o do banco de dados. Em seguida é definido o IP da máquina de onde serão destinados os ataques, para que seja possível fazer a categorização dos dados utilizados no treinamento dos modelos e os respectivos testes.

Na sequência o programa entra em uma estrutura de repetição *for* de captura contínua de pacotes de rede. Nela, o primeiro passo é filtrar por pacotes do tipo TCP, suportado no nosso processo, e em seguida, através de uma função específica, extrair os dados necessários para continuidade do processo, sendo eles: IP de destino e origem, porta de destino e origem, *timestamp*, valor booleano para as *flags* ACK e SYN e tamanho da janela de dados em *bytes*. Feito isso, os dados são inseridos no banco de dados relacional na tabela "*captures*", ação que retorna um ID, o qual é inserido na fila de processamento *Redis*, também denominada como "*captures*".Concluídos todas as etapas sem erro, é iniciado a análise do próximo pacote.

3.2.3 Aquisição dos dados para treinamento do modelo

Para adquirir os dados de treinamento, foi preciso desativar o serviço de classficação dos dados, visto que ainda não havia um modelo treinado para isso. Em seguida, foram definidas 6 réplicas do serviço de processamento dos dados, para não criar um gargalo enorme na fila de processamento.

Feito isso, passamos para a simulação de cada um dos três tipos de ataque definidos: *Portscan*, *Bruteforce* e *Denial of Service*, e também de tráfego de rede benigno. Cada tipo de ataque foi simulado em execuções isoladas, misturando apenas dados do ataque em

questão com tráfego seguro. Ao fim foram capturadas 698311 tentativas de requisição, sendo 354164 benignas e 344147 malignas.

Cada ataque foi simulado utilizando um *framework Linux*, a fim de simplificar o processo de simulação. O ataque do tipo *Portscan* foi obtido atráves da ferramenta *nmap* e configurado para varrer todas as portas da máquina atacada. O *Denial of Service* utilizou a biblioteca *HPING3* para realizar as requisições em massa na porta 8000, única porta aberta para conexão na máquina de destino. Os ataques de *Bruteforce* foram executados através da ferramenta *hydra* e do repositório de senhas vazadas *rockyou*, definindo usuário estático como "admin", este ataque é o único que utiliza de forma clara uma rota da aplicação hospedada na máquina de destino. Realizando inúmeras requisições na rota */login*, até que retorne sucesso na autenticação.

Para executar os ataques de *portscan*, *bruteforce* e *DoS* foram executados os seguintes comandos, por ordem:

1. nmap -p- <HOSTNAME_DESTINY>
2. hydra -l admin -P rockyou.txt <HOSTNAME_DESTINY> http-post-form "/login:username=^USER^&password=^PASS^:Invalid credentials-s 8000 -f
3. hping3 -S <IP_SOURCE> -a <HOSTNAME_DESTINY> -p 8000 –flood

Todos os ataques rodam em paralelo a requisições benignas enviadas pelo serviço dedicado a essa função. Como o tráfego dos ataques por muitas vezes sobrecarrega o serviço da aplicação e impede o tráfego benigno, foi necessário manter algumas seções de aquisição somente com esse tipo de requisições para completar a base de dados com uma proporção próxima entre os diferentes tipos.

Os serviços destinados ao processamento dos dados capturados e obtenção dos dados do modelo foram desenvolvido para capturar 10 variáveis envolvendo a requisição analisada e o histórico de tráfego das últimas 20 mil conexões entre os *hosts* envolvidos.

As variáveis obtidas para uso no modelo foram:

1. *Mean Window Size Forward*: Tamanho médio das janelas do pacotes enviados;
2. *Mean Window Size Backward*: Tamanho médio das janelas do pacotes recebidos;
3. *ACK Count*: *Flag* ACK presente no pacote;
4. *SYN Count*: *Flag* SYN presente no pacote;
5. *Forward Packets/s*: Quantidade média de pacotes enviados por segundo;
6. *Backward Packets/s*: Quantidade média de pacotes recebidos por segundo;

7. *IAT Max*: Tempo máximo entre dois pacotes diferentes;
8. *IAT Mean*: Tempo médio entre dois pacotes diferentes;
9. *IAT Min*: Tempo mínimo entre dois pacotes diferentes;
10. *Ports Number*: Somatório do número de portas diferentes utilizadas na conexão.

Os parâmetros escolhidos foram baseados nos dados coletados no CICIDS desenvolvido por Sharafaldin, Lashkari, Ghorbani et al. (2018) e destacados por Pelletier e Abualkibash (2020), e também em um conhecimento empírico relacionado aos tipos de ataques que seriam simulados, como a escolha do parâmetro *Ports Number* que pode ressaltar fortemente a ocorrência de um ataques de *portscan* a medida que o valor aumenta.

Ao fim de cada uma das rodadas de coleta de dados, é feito a extração de um arquivo CSV da tabela "*alerts*"no banco de dados e armazenado em um repositório *GIT* na plataforma *Github*, que serão consumidos na etapa de pré-processamento dos dados.

3.2.4 Pré-processamento dos dados

Para realizar o pré-processamento e limpeza dos dados foi utilizado a linguagem *Python* embarcado na ferramenta *Google Colab*, construída especialmente para uso em desenvolvimento de projetos de dados, além disso foi necessário utilizar as bibliotecas *pandas*, *numpy* e *sklearn*.

Inicialmente, os dados são agrupados em um único *dataframe pandas*, já que os dados são disponibilizados em 11 arquivos do tipo CSV diferentes. Em seguida é feita a seleção dos 10 atributos utilizados para o desenvolvimento dos modelos de aprendizado de máquina e dos rótulos de cada item, mantendo assim as seguintes colunas: 'mean_win_fwd', 'mean_win_bwd', 'ack_count', 'syn_count', 'fwd_pck', 'bwd_pck', 'iat_max', 'iat_mean', 'iat_min', 'ports_number' e 'label'.

Em seguida é elaborado um pequeno relatório contendo informações relacionadas aos dados de cada uma das colunas mantidas, são exibidos número total de dados válidos, valor médio, desvio padrão, valores máximos e mínimos de cada uma, e os quartis, como mostra a figura 12. O intuito do relatório é dar uma visão macro de cada coluna e apoiar a tomada de decisão na próxima etapa do pré-processamento do conjunto.

Em seguida, para atender as necessidades de desenvolvimento das redes neurais, que trabalham com saídas numéricas, é utilizado o módulo de *LabelEncoder* do pacote *sklearn* para transformar os rótulos '*BENIGN*' e '*MALIGN*' em 0 e 1, respectivamente.

Feito todo o pré-processamento dos dados, que será atribuído para todos os modelos treinados, devemos montar os conjuntos de dados em treinamento de teste, para isso foi utilizado a função *train_test_split* do pacote *sklearn*, no qual parâmetro *test_size* foi

	mean_win_fwd	ack_count	syn_count	fwd_pck	iat_max	iat_mean	iat_min	ports_number
count	698311.000000	698311.000000	698311.000000	698311.000000	698311.000000	698311.000000	698311.000000	698311.000000
mean	27996.748552	0.817392	0.209577	2.525823	1301.603447	0.562582	0.000001	1872.614995
std	31506.141350	0.386345	0.407007	2.770106	2142.435439	1.622357	0.001197	3183.828346
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	1.000000
25%	0.000000	1.000000	0.000000	0.990000	45.000000	0.250000	0.000000	1.000000
50%	1024.000000	1.000000	0.000000	1.470000	876.000000	0.370000	0.000000	2.000000
75%	64190.000000	1.000000	0.000000	2.770000	1510.000000	0.620000	0.000000	2909.000000
max	65160.000000	1.000000	1.000000	38.000000	11128.000000	108.630000	1.000000	10000.000000

Figura 12 – Informações de detalhamento das colunas do conjunto de dados obtido.

definido como 0.3, o qual atribuí 70% dos dados para treinamento de 30% para testes. Foi escolhido este método de separação dos dados pois ele garante a participação de todas os rótulos em ambos os conjuntos de dados, já a proporção dos dados treinamento/teste foi escolhida de forma arbitrária. Por fim, os dados são exportados em quatro arquivos CSV, os quais serão utilizados posteriormente para treinamento dos modelos.

3.3 Treinamento dos modelos *Decision Tree* e *Random Forest*

Para que seja possível realizar comparações de desempenho com as redes neurais, foram implementados dois modelos de aprendizado de máquina: *Decision Tree* e *Random Forest*. Ambos os modelos foram desenvolvidos em *Python* e utilizando a ferramente *Google Colab* e as bibliotecas *pandas*, *sklearn*, *numpy*, matplotlib e seaborn.

O primeiro passo é a importação dos dados tratados na etapa anterior, para isso utilizou-se o *pandas* que realiza a leitura dos 4 arquivos CSV e os armazena em variáveis do tipo *dataframe*, denominados *dtX_train, dtX_test, dtY_train* e *dtY_test*.

Além disso, foram treinados modelos com 10 parâmetros de entrada, contendo todos os dados coletados, e modelos com 8 parâmetros, nos quais foram removidos as colunas de dados do tipo *Backward*. Esta variação nos parâmetros de entrada tem o intuito de validar o desempenho apresentado pelo modelo no uso destes dados, visto que eles possuem valores semelhantes aos do tipo *Forward*, o mesmo se deu ao treinar as redes neurais.

O modelo de *Decision Tree* foi implementado atráves da biblioteca *Sklearn*, utilizando o módulo *DecisionTreeClassifier*, que disponibiliza um modelo de árvore de decisão previamente configurado, sendo necessário apenas realizar as funções de treinamento e validação, já que optou-se por manter os parâmetros padrões do módulo.

Desta forma, com o modelo já criado realizamos o treinamento utilizando os dados dos conjuntos *dtX_train* e *dtY_train*, e após treinado fazemos a predição dos dados contidos no *dtX_test*, resultando no conjunto de rótulos que será comparado com o *dtY_test*, e assim obtemos as métricas do modelo. Para isso utilizamos alguns módulos do pacote *sklearn.metrics*, o *accuracy_score* e o *confusion_matrix*, que retornam a acurácia e

a matriz de confusão (que por si só possui diversas métricas incorporadas). Para concluir utilizamos as bibliotecas *seaborn* e *matplotlib* para obter a imagem da matriz de confusão do modelo.

A metodologia para o modelo de *Random Forest* é similar, primeiramente fazemos a importação do módulo *RandomForestClassifier* do *sklearn*. Porém, para este iremos definir o parâmetro *max_depth* igual a 2, esse atributo define a profundidade máxima de cada árvore que será criada na floresta randômica, e foi escolhido de forma aleatória. Os demais passos da criação da *Random Forest* são idênticos aos descritos no modelo de *Decision Tree*.

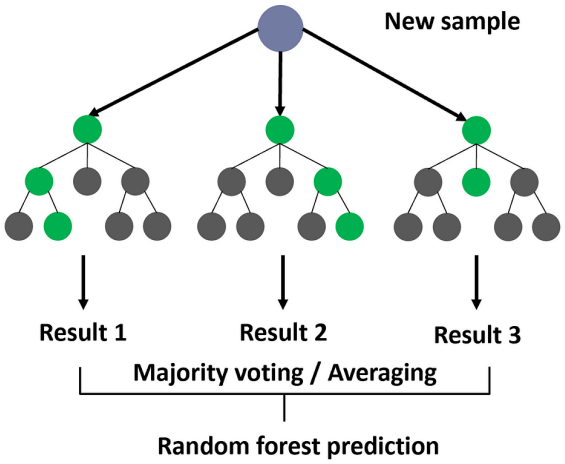

Figura 13 – Representação gráfica do algoritmo de *random forest*. Fonte: Yehoshua (2023)

3.4 Treinamento de Rede Neural Artifical Binária

A rede neural abordada neste trabalho será um modelo de classificação binária, mantendo os 2 rótulos originais do conjunto de dados. Para construir está rede neural será utilizada a linguagem *Python* e o *Google Colab*, e para criar a RNA utilizaremos o *Tensorflow*, que é uma biblioteca do *Google* feita para trabalhar com modelos de aprendizado profundo. Além disso, serão utilizados *pandas*, *sklearn*, *numpy*, matplotlib e seaborn.

Os modelos criados a partir do *Tensorflow* exigem uma etapa de pré-processamento adicional. Inicialmente fazemos a importação dos 4 conjuntos de dados e em seguida a transformação dos *dataframes pandas* obtido para uma estrutura de vetores *numpy* utilizando a função *to_numpy*, a transformação deverá ser aplicada nos quatro conjuntos de dados importados.

Na etapa de treinamento das redes neurais foram avaliadas 6 diferentes composições de redes, para que possamos comparar os resultados e encontrar a melhor escolha de

arquitetura da rede. Para isso, foram utilizadas redes com 10 parâmetros de entrada, com variações de 128, 64, 32 e 16 nós na camada escondida, e redes com 8 (sem dados de *Backward*), e suas variações com 128 e 64 nós.

Feito todo o pré-processamento necessário, iniciamos o desenvolvimento da rede neural, para isso o primeiro passo é definir a estrutura de camadas, utilizaremos o módulo *keras.layers* embarcado no *Tensorflow* para isso. Começando pela camada de entrada, criamos um *layer* do tipo *Input* com a quantidade de nós igual a de parâmetros de entrada da rede (8 ou 10), que equivale a quantidade de atributos que nosso modelo receberá para usar nas predições, logo criamos um nó receptor para cada um deles.

Em seguida criamos uma única camada oculta utilizando um *layer* do tipo *Dense* com 64 ou 128 nós e função de ativação *'Relu'*, ambos os parâmetros foram selecionados mediante comportamento em testes, sendo essa combinação a que apresentou melhor desempenho entre as demais testadas. A camada de saída é a última a ser definida e também será do tipo *Dense*, porém para a camada de saída teremos 2 nós, respeitando a quantidade de possíveis rótulos, já que o valor de cada nó representa a probabilidade de que aquele seja o rótulo correto, e para ativação da camada foi selecionada a função *'Softmax'*.

Com todas as camadas bem definidas é feita a compilação do modelo que define alguns parâmetros em relação ao treinamento. Neste casos os parâmetros foram definidos da seguinte forma: *optimizer* igual a 'adam', que utiliza o algoritmo de Adam para fazer a convergência da rede, *loss* igual a *'sparse_categorical_crossentropy'*, que calcula a entropia cruzada entre rótulos e predições, por fim, *metrics* como *'accuracy'*.

Em seguida foi definido um *Early Stopping* para o treinamento da rede, esse recursos permite que a rede interrompa o treinamento caso as perdas aumentem consecutivamente e restaura a época que apresentou o melhor desempenho no treinamento. O *Early Stopping* foi implementado utilizando o módulo *callbacks* do *keras* e utiliza acurácia como métrica analisada para avaliar a rede treinada, permitindo até 5 pioras consecutivas no desempenho antes de interromper o processo. Caso não aconteça nenhuma interrupção, o treinamento acontece até o número máximo de épocas definido.

O próximo passo é fazer o treinamento do modelo, para isso foi definido o número máximo de 500 épocas, através do parâmetro *epochs*, e o uso do *Early Stopping* no parâmetro *callback*, visando concluir o processo em um tempo considerável, já que existiam limitações de recursos de máquina. Além disso, o parâmetro 'shuffle' foi definido como verdadeiro, fazendo com que os dados sejam apresentados para a rede em ordens diferentes em cada época, para garantir que a ordem dos dados não seja determinante no resultado final do modelo.

Realizados todos estas etapas, podemos utilizar nosso modelo para predição dos

conjuntos de teste e validar seu desempenho através das mesmas métricas e ferramentas trabalhadas com os modelos clássicos descritos em 3.3. É importante ressaltar que por se tratar de um rede neural de 2 nós de saída, o resultado de uma predição é um vetor de tamanho 2, contendo valores entre 0 e 1, que correspondem a probabilidade de cada rótulo ser a saída real, para tornar esse vetor em um valor único utilizamos a função *numpy.argmax* que retorna o *index* do argumento com maior valor, logo, o mais provável entre as 2 opções.

3.5 Sistema de monitoramento de detecção de ataques

A última etapa deste trabalho consistiu na elaboração de um sistema de monitoramento do tráfego de rede e detecção de ataques em tempo real através dos modelos obtidos nas etapas anteriores. Para isso, fazemos a exportação do modelo *tensorflow* obtido através do treinamento na etapa anterior e carregamos no sistema de simulação descrito na primeira etapa. Feito isso, podemos habilitar o serviço de análise no contêiner *docker*, que será responsável por utilizar a rede neural para classificar as conexões. E para dar maior vazão aos dados, definimos o número de réplicas para 3, sendo assim três máquinas processam os dados da fila *Redis 'alerts'* e executam a classificação.

Por fim, simulamos novamente todo os três ataques e coletamos os dados para que fossem submetidos a análises dos resultados e assim podemos avaliar o desempenho do modelo desenvolvido no ambiente real. Diferente da forma como foi realizada a coleta de dados para treinamento, aqui simulamos um ambiente produtivo, no qual o histórico ficaria armazenado por tempo indeterminado e os ataques acontecem de forma sequencial.

4 Resultados

4.1 Resultados obtidos com CICIDS2017

No inicio do trabalho foram desenvolvidos três modelos de aprendizado de máquina (RNA, *Decision Tree* e *Random Forest*) utilizando os dados do conjunto CICIDS2017. Estes modelos chegaram a apresentar resultados interessantes, onde já era possível entender o comportamento de cada uma das técnicas escolhidas neste trabalho.

O modelo *Decision Tree* apresentou ótimo desempenho, com acurácia de 99,80% e *F1-score* de 99,49%, como mostra matriz de confusão na figura 14.

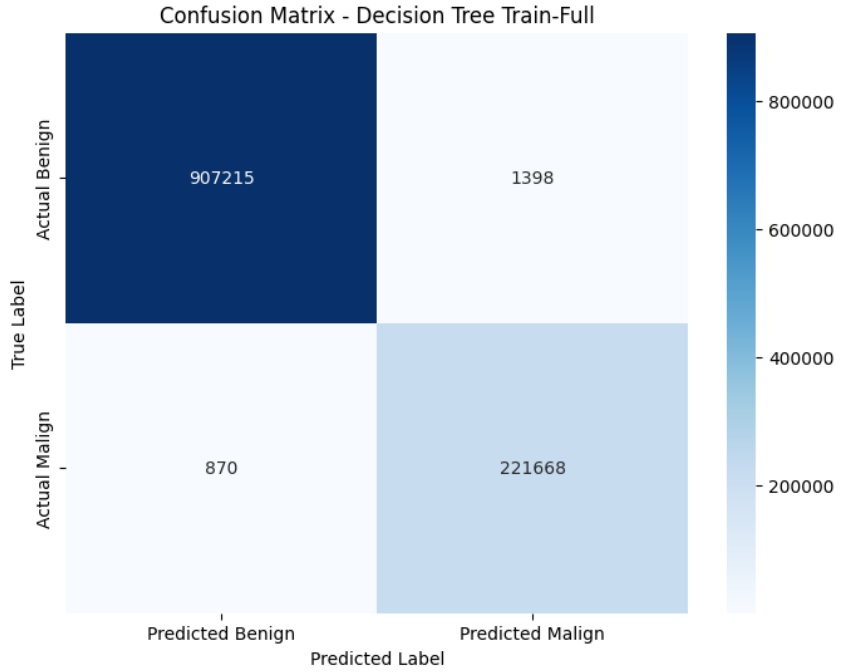

Figura 14 – Matriz de Confusão para *Decision Tree* com dados do CICIDS2017. Fonte: Autor

Ao treinar o modelo de *Random Forest*, inicialmente definindo o parâmetro *max_depth* como 2, foram obtidos resultados abaixo do esperado, desta forma, realizamos novos treinamentos variando o parâmetro e avaliando os resultados, que foram melhorando significativamente. A exatidão do modelo variou de 86,01%, com *max_depth* igual a 2, até 99,54%, com o parâmetro igual a 12, como detalha a tabela 1.

Por fim, foi realizado o treinamento do modelo de Rede Neural Artificial, o qual apresentou bom resultados, próximos aos do modelo de *Decision Tree*. A RNA desenvolvida contou com 128 neurônimos na camada escondida e a função de ativação 'Relu'. O modelo teve acurácia de 97,35% e *F1-score* de 93,34%, como mostra a figura 15.

Tabela 1 – Métricas do treinamento dos modelos de *Random Forest* com variação do parâmetro *max_depth* utilizando base CICIDS2017

max_depth	Acurácia	Precisão	Recall	F1
2	0,8601	1	0,2887	0,4481
3	0,8606	1	0,2915	0,4514
4	0,9385	0,9972	0,6894	0,8152
5	0,9525	0,9965	0,7611	0,8631
8	0,9878	0,9895	0,9479	0,9683
12	0,9954	0,9903	0,986	0,9882

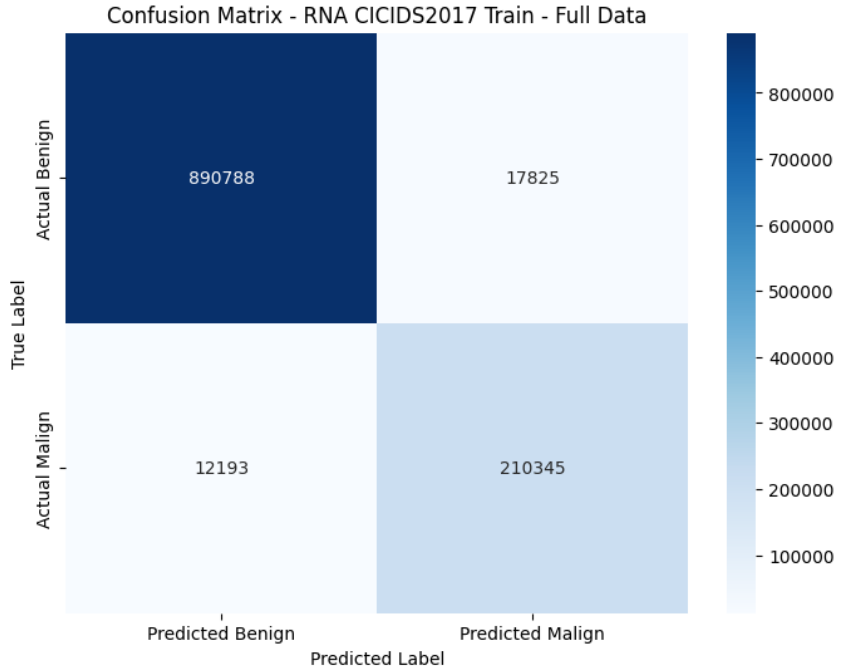

Figura 15 – Matriz de Confusão para Rede Neural Artificial com dados do CICIDS2017. Fonte: Autor

4.2 Treinamento dos Modelos

Ao longo do treinamento de cada um dos modelos desenvolvidos foram validadas várias hipóteses para encontrar a melhor combinação de parâmetros para atender ao problema apresentado, como a seleção dos dados de entrada e a quantidade de nós na camada escondida de um modelo de rede neural artificial. Todos os modelos foram submetidos as métricas de: acurácia, precisão, *recall* e *f1-score*, sendo consideradas as métricas mais valiosas no objetivo proposto para os modelos. Todas os dados apreentados foram obtidos tráves de código *Python* e utilizando o módulo de *metrics* do *framework Sklearn* e a biblioteca *Matplotlib* para geração de gráficos.

As métricas foram escolhidas visando representar parâmetros importantes para o contexto. A acurácia faz uma análise dos acertos de forma geral, sendo obtida através da razão entre o número de acertos e do total de predições. A precisão representa o

desempenho do modelo em evitar resultados falso-positivos, sendo dado pela razão das predições verdadeiro-positivo e todas as predições positivas (verdadeiras e negativas). o *recall* é uma das métricas mais importantes dentro do contexto deste trabalho e representa a capacidade do modelo em não obter falso-negativos em suas predições, e é obtido pela divisão entre os verdadeiro-positivos e todos os itens positivos(verdadeiro-positivo e falso-negativo). Por fim, o *f1-score* é uma métrica que combina as duas anteriores em uma única medida.

A seguir serão detalhados os dados de desempenho de cada um dos modelos de acordo com os parâmetros selecionados no treinamento e quanto ao conjunto de dados utilizado na predição, detalhando o resultado dos dados de teste e dos dados obtidos em simulação com modelo.

4.2.1 *Decision Trees*

Durante o treinamento dos modelos de árvore de decisão foram implementadas duas opções de configuração dos dados, 10 e 8 parâmetros de entrada, descritos em 3.2.3, e avaliado o desempenho de cada uma delas. Em ambas foram apresentados ótimos resultados em todas as métricas avaliadas, como mostram as matrizes de confusão, havendo um desempenho ligeiramente melhor no modelo com 10 parâmetros de entrada.

O primeiro modelo de árvore de decisão treinado, com 10 parâmetros de entrada, apresentou um ótimo desempenho no treinamento, com a acurácia e demais métricas avalidas com valor muito próximo do máximo. No segundo modelo, com a remoção de 2 parâmetros de entrada no treinamento da árvore, também foi observado um resultado próximo de 1 para as métricas avaliadas.

Tabela 2 – Métricas do treinamento dos modelos de árvore de decisão

Modelo	Acurácia	Precisão	Recall	F1
Decision Tree Train-Full	1	1	1	1
Decision Tree Train-Reduced	1	1	0,9999	1

4.2.2 *Random Forest*

Os modelos de *Random Forest* seguiram as mesmas variações das árvore de decisão, sendo avaliados modelos com 10 e 8 parâmetros de entrada. Foram apresentados bons resultados, com valores entre 1 e 0.9 para todas as métricas observadas, e uma variação de menor que 1 ponto percentual entre as duas variações, sendo o modelo com 10 parâmetros aquele que apresentou melhores resultados na maioria das métricas, porém, o modelo reduzido conseguiu desempenhar melhor na precisão, que demonstra a capacidade do modelo em apresentar falsos positivos.

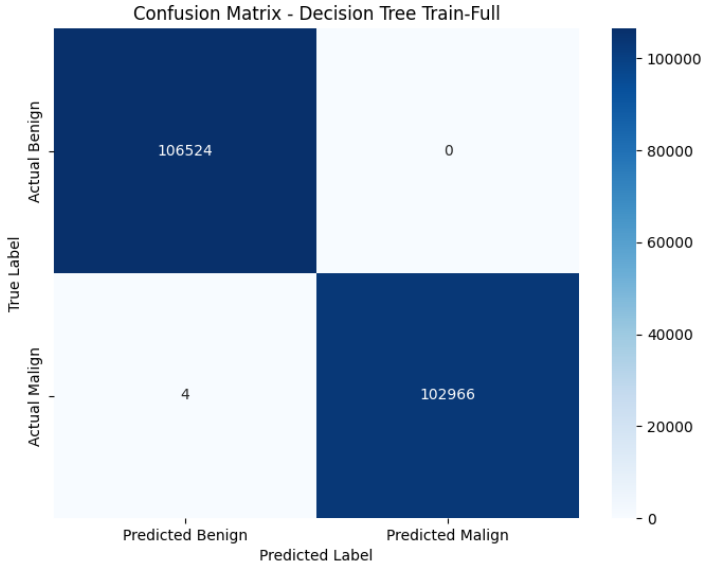

Figura 16 – Matriz de confusão de árvore de decisão treinada com 10 parâmetros de entrada. Fonte: Autor

Figura 17 – Matriz de confusão de árvore de decisão treinada com 8 parâmetros de entrada. Fonte: Autor

Tabela 3 – Métricas do treinamento dos modelos *Random Forest*

Modelo	Acurácia	Precisão	Recall	F1
Random Forest Train-Full	0,9727	0,9973	0,9469	0,9715
Random Forest Train-Reduced	0,9678	0,9978	0,9364	0,9662

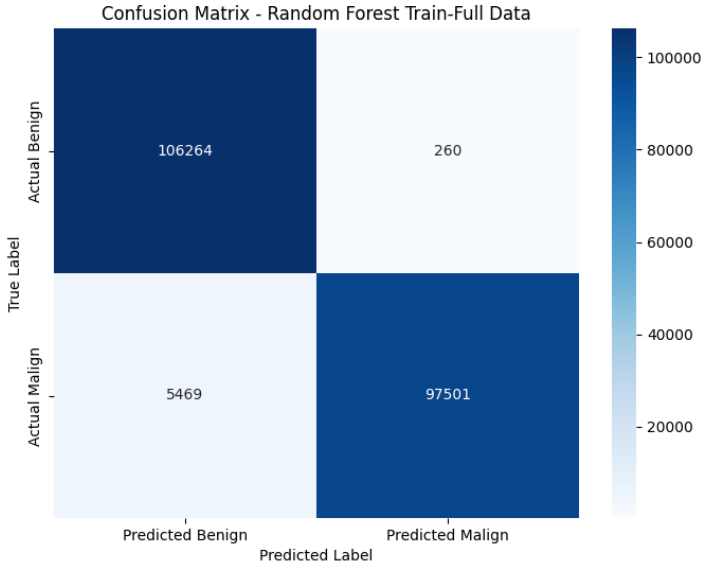

Figura 18 – Matriz de confusão de *Random Forest* treinada com 10 parâmetros de entrada. Fonte: Autor

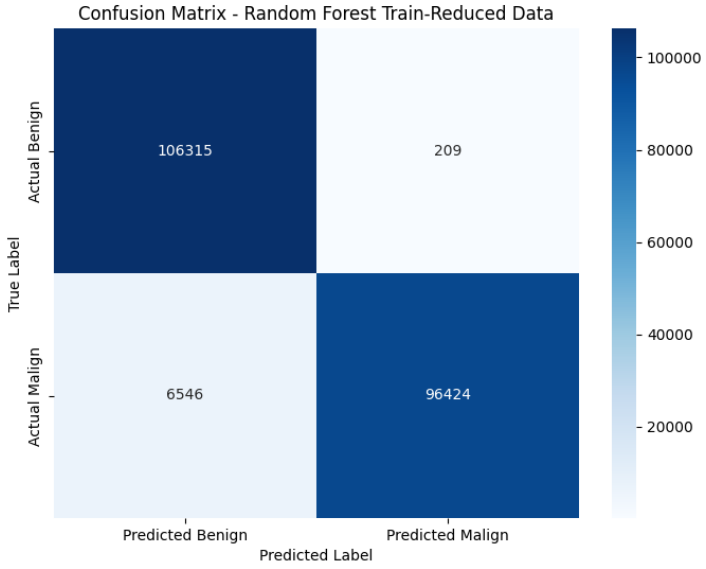

Figura 19 – Matriz de confusão de *Random Forest* treinada com 8 parâmetros de entrada. Fonte: Autor

4.2.3 Rede Neural Artificial

Foram desenvolvidos 6 tipos de modelos de redes neurais, variando os parâmetros de entrada e as especificações da camada escondida do modelo. As variações com 10 e 8 parâmetros de entrada intercalando entre 128 e 16 neurônios na camada escondida. A cada execução, mesmo que fixado os parâmetros de aleatoriedade, os resultados entre modelos com mesma configuração divergiam bastante. Devido a isso, foram executados 10 séries de

treinamento e coletados todos as métricas para análise (7)(8). Sendo assim, foram gerados 60 modelos para comparação de resultados.

Tabela 4 – Valores médios apresentados por cada modelo de *Redes Neurais Artificiais*

	Modelo	F-Score	Acurácia	Precisão	Recall
0	Full128	0.988430	0.988790	0.998780	0.978380
3	Full64	0.984350	0.984840	0.994750	0.974460
2	Full32	0.980560	0.981780	0.995640	0.967050
1	Full16	0.980040	0.981090	0.996550	0.964700
5	Reduced64	0.924460	0.917420	0.926810	0.936470
4	Reduced128	0.897650	0.888180	0.902650	0.914670

Devido a essa variação nos resultados, optamos por avaliar a melhor configuração de modelo à partir da média dos resultados (4), que elencou o modelo de 10 parâmetros de entrada e 128 neurônios como obtendo o melhor desempenho entre as demais configurações testadas. De outro lado, os modelos reduzidos, com apenas 8 parâmetros de entrada, apresentaram o pior desempenho. O modelo reduzido com 128 neurônios na camada escondida apresentou o pior desempenho entre todos.

O modelo com o melhor desempenho obteve um valor de *F1-Score* de 0.9988, em contrapartida o pior desempenho obteve 0.7403 para a mesma métrica. A Matriz de confusão de cada um destes modelos pode ser analisada nas figuras 20 e 21, respectivamente. Podemos notar que o modelo com pior desempenho teve todas as suas predições com erros do tipo falso-positivo, que são considerados menos nocivos dentro do nosso contexto, onde um alerta falso é menos comprometedor do que a omissão de um caso de ataque.

Importante ressaltar o desempenho dos modelos com 10 parâmetros de entrada e 16 neurônios na camada escondida, com resultados que se assemelham bastante aos melhores modelos, porém, com um custo computacional muito inferior, o que pode justificar o uso destes modelos em sistemas que permitem essa menor acurácia e recursos de *hardware* limitados.

Os resultados obtidos pelo modelo de rede neural foi ótimos e se assemelharam aos dos outros modelos, com boa taxa de acertos e número baixo de erros do tipo falso-negativo, que se mostram mais danosos para nossa aplicação.

Tabela 5 – Métricas do treinamento da melhor série entre os modelos de Redes Neurais Artificiais

	Série	Modelo	F-Score	Acurácia	Precisão	Recall
42	7	Full128	0.998800	0.998800	0.998100	0.999500

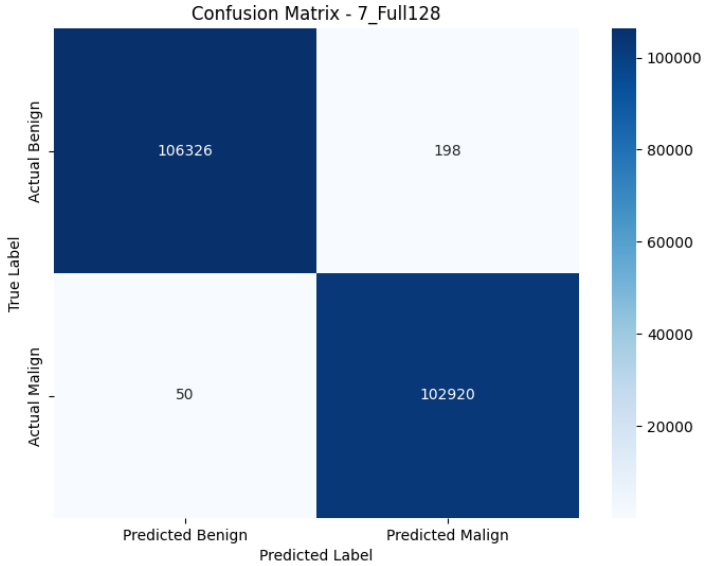

Figura 20 – Matriz de confusão do melhor modelo de RNA com 128 nós e 10 parâmetros de entrada. Fonte: Autor

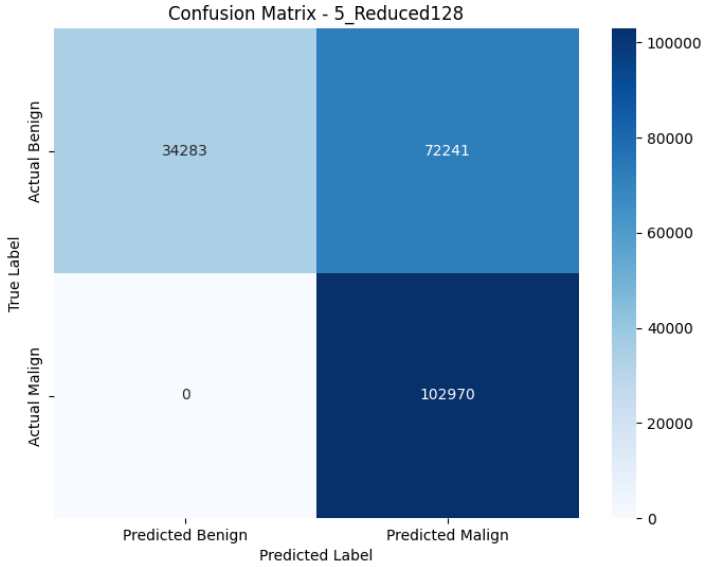

Figura 21 – Matriz de confusão do pior modelo de RNA com 128 nós e 8 parâmetros de entrada. Fonte: Autor

4.3 Validação dos modelos em simulação

O processo foi implementado utilizando o modelo de rede neural que apresentou o melhor resultado nos testes, com 128 neurônios da camada escondida e 10 parâmetros de entrada. Para efeito de comparação, os dados finais foram extraídos e reprocessados utilizando as melhores versões dos modelos de *Decision Tree* e *Random Forest*, sendo o

modelo de árvore de decisão o que apresentou o melhor resultado entre os avaliados.

Tabela 6 – Métricas por modelo de dados coletados em ambiente produtivo

Modelo	F-Score	Acurácia	Precisão	Recall
Decision Tree Prod-Full	0.804000	0.803300	0.796800	0.811300
Best RNA Full128	0.508200	0.654800	0.871300	0.358700
Random Forest Prod-Full Data	0.332700	0.590800	0.880100	0.205100

Através das matrizes de confusão podemos avaliar no detalhe o desempenho de cada um dos três modelos utilizando dados coletados na simulação. Nas figuras 22 e 24 é possível visualizar um grande desequilíbrio no resultado dos respectivos modelos, que tiveram um número superior de falsos negativos (falso benigno). Já o modelo *Decision Tree*, apresentou resultados mais equilibrados, como visto na figura 23.

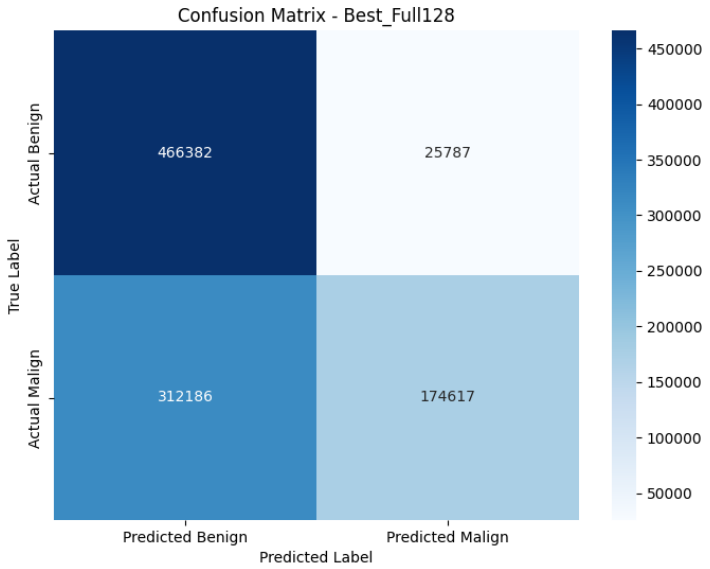

Figura 22 – Matriz de confusão do modelo de RNA com 128 nós e 10 parâmetros de entrada executando em simulação. Fonte: Autor

É notável a perda de performance ao utilizar os modelos desenvolvidos em um ambiente produtivo. O principal motivo apontado se dá pela diferenciação do histórico de requisições entre as coletas de dados para teste e o fluxo em simulação, no segundo os ataques são executados de forma sequencial, desta maneira o ataque executado anteriormente, pela mesma fonte (mesmo IP), acaba influenciando o histórico de requisições e adicionando anomalias nas variáveis que envolvem análise de várias requisições no mesmo sentido (cálculos de média, por exemplo).

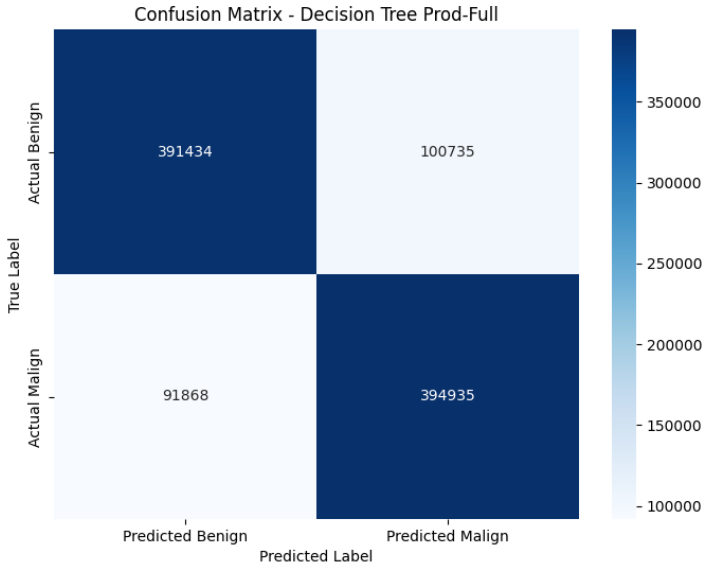

Figura 23 – Matriz de confusão do modelo de árvore de decisão de 10 parâmetros de entrada em simulação. Fonte: Autor

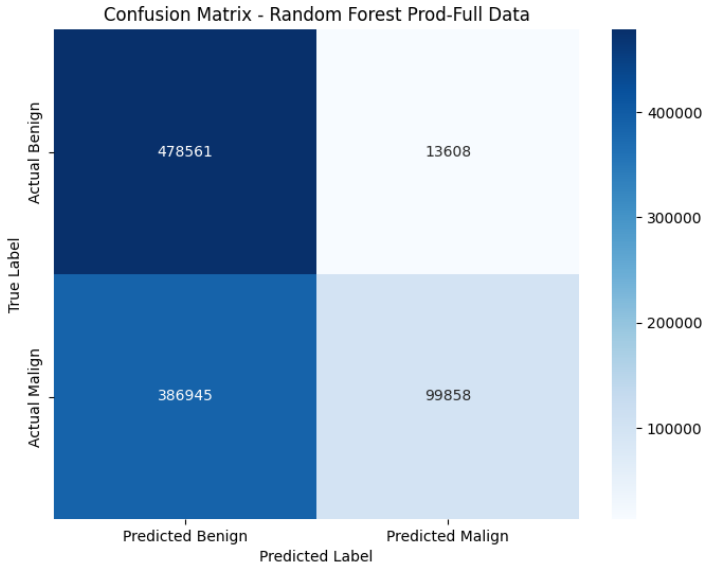

Figura 24 – Matriz de confusão do modelo de floresta randômica de 10 parâmetros de entrada em simulação. Fonte: Autor

5 Conclusões e Sugestões para Trabalhos Futuros

O trabalho avaliou o desempenho e a dificuldade de implementação de modelos de redes neurais artificiais na detecção de ataques cibernéticos comuns. Além disso, buscou-se comparar as RNA com outros modelos clássicos, que exigem custos de treinamento e processamento reduzidos. Visando encontrar estas respostas, obtemos o seguinte resultado: sim, redes neurais podem apresentar boas métricas na classificação de tráfego de rede, porém, o uso de um modelo de menor custo, como *Decision Trees*, pode trazer mais benefícios para este uso, visto que funcionam muito bem e requerem menor poder de processamento.

A começar pela escolha do ambiente para simulação. Os contêineres *Docker* se mostraram como uma ótima opção, sendo possível controlar os recursos de máquina de maneira eficaz, adicionar e remover funções do escopo de forma simplificada, e principalmente, por garantir que o ambiente esteja dentro dos parâmetros estabelecidos independente de qualquer configuração ou mudança de estado da máquina utilizada como hospedeiro (*host*). Desta maneira é possível que qualquer um consiga baixar os códigos utilizados nas implementações deste trabalho e utilize em sua própria máquina, sem riscos de incompatibilidade. Desta forma o ambiente escolhido foi fator principal para reprodutibilidade deste trabalho.

Em relação ao desempenho dos modelos desenvolvidos, pudemos observar que a rede neural utilizada apresentou bom desempenho diante dos dados de teste, atingindo a ótima acurácia de 99,88% e um *recall* de 99,95%. Porém, ao utilizar este mesmo modelo em um ambiente produtivo, obtemos resultados abaixo, porém, promissores, com uma acurácia de 65,48%.

O destaque em relação ao desempenho ficou para o modelo *Decision Tree* que apresentou os melhores resultados tanto nos testes quanto no ambiente produtivo, com acurácia e *recall* próximos de 1 no treinamento, e 80,40% no uso na simulação, sendo o último um valor muito superior aos outros modelos. Em contrapartida, o modelo de *Random Forest*, apresentou os piores resultados, alcançando uma acurácia de 97,27% no treinamento, e um resultado muito abaixo na simulação, chegando a apenas 59,08%.

O trabalho apresentou algumas limitações relacionados a conclusão do estudo em tempo hábil e aos recursos disponíveis para desenvolvimento, ambas listadas abaixo:

1. Limitação de recursos de máquina que permitisse a criação de um número maior de réplicas dos contêineres responsáveis pelas análises de dados do tráfego.

2. Variedade de ataques simulados reduzidos.

Diante das limitações apresentadas, surgem algumas recomendações para trabalhos futuros:

1. Explorar diferentes métodos e combinações de atributos de entrada do modelo.
2. Utilizar técnicas de validação cruzada para avaliar a capacidade de generalização do modelo.
3. Implementação de rótulos relacionados ao tipo de ataque, a fim de conseguir informar ao usuário com mais clareza o que se passa na aplicação monitorada.
4. Simulação de novos tipos de ataques, aumentando a lista atual de 3 tipos, buscando sempre manter o balanceamento das classes.
5. Utilizar todos os 3 modelos desenvolvidos no ambiente produtivo e avaliar o consumo de processamento de cada um deles.

Por fim, o trabalho atingiu o objetivo esperado, sendo possível visualizar o comportamento de cada um dos modelos avaliados e abrindo portas para melhorias e implementações no futuro. A arquitetura escolhida para implementação das ferramentas também se mostrou eficiente e de fácil escalabilidade, permitindo o uso em sistemas diversos, com diferentes capacidades de processamento, e demonstrando uma grande potencial para aplicações reais sem elevar de forma considerável os custos.

Referências

AMAZON Machine Learning: Guia do desenvolvedor. https://docs.aws.amazon.com/pt_br/machine-learning/latest/dg/machinelearning-dg.pdf. Acessado em: 2024-06-10. Citado 1 vez na página 21.

BRAGA, Antônio de Pádua; LUDERMIR, Teresa Bernarda; CARVALHO, André Carlos Ponce de Leon Ferreira. Redes neurais artificiais: teoria e aplicações, 2000. Citado 2 vezes na página 21.

BRANCO, Henrique. *Overfitting e underfitting em Machine Learning*. 2020. Acessado em: 26 de setembro de 2024. Disponível em: https://abracd.org/2020/08/21/overfitting-e-underfitting-em-machine-learning/. Citado 1 vez na página 24.

BRASIL, BBC News. *Ataque Cibernético Paralisa Maior Rede de Gasodutos dos EUA*. 2021. Acessado em: 27 de setembro de 2024. Disponível em: https://www.bbc.com/portuguese/internacional-57055618. Citado 1 vez na página 12.

CH, Rupa et al. Computational system to classify cyber crime offenses using machine learning. *Sustainability*, MDPI, v. 12, n. 10, p. 4087, 2020. Citado 3 vezes nas páginas 15, 16.

CINQUE, Marcello; COTRONEO, Domenico; PECCHIA, Antonio. Challenges and directions in security information and event management (SIEM). In: IEEE. 2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). 2018. P. 95–99. Citado 1 vez na página 15.

COHEN, Fred. Computer viruses: theory and experiments. *Computers & security*, Elsevier, v. 6, n. 1, p. 22–35, 1987. Citado 1 vez na página 11.

FÜRNKRANZ, Johannes; GAMBERGER, Dragan; LAVRAČ, Nada. *Foundations of rule learning*. Springer Science & Business Media, 2012. Citado 1 vez nas páginas 18, 19.

GEEKS, Geeks For. *Random Forest Algorithm in Machine Learning*. 2025. Acessado em: 16 de março de 2025. Disponível em: https://www.geeksforgeeks.org/random-forest-algorithm-in-machine-learning/. Citado 0 vez na página 20.

HAGAN, Martin T; DEMUTH, Howard B; BEALE, Mark. *Neural network design*. PWS Publishing Co., 1997. Citado 6 vezes nas páginas 22, 23.

HO, Tin Kam. Random decision forests. In: IEEE. PROCEEDINGS of 3rd international conference on document analysis and recognition. 1995. v. 1, p. 278–282. Citado 2 vezes nas páginas 19, 20.

KPMG. *KPMG global tech report 2023*. 2023. Acessado em: 27 de setembro de 2024. Disponível em: https://kpmg.com/xx/en/our-insights/ai-and-technology/kpmg-global-tech-report-2023.html. Citado 1 vez na página 12.

LANGNER, Ralph. Stuxnet: Dissecting a cyberwarfare weapon. *IEEE security & privacy*, IEEE, v. 9, n. 3, p. 49–51, 2011. Citado 1 vez na página 11.

LEK, Sovan; GUÉGAN, Jean-François. Artificial neural networks as a tool in ecological modelling, an introduction. *Ecological modelling*, Elsevier, v. 120, n. 2-3, p. 65–73, 1999. Citado 2 vezes na página 23.

MOUKAFIH, Nabil; ORHANOU, Ghizlane; EL HAJJI, Said. Neural network-based voting system with high capacity and low computation for intrusion detection in SIEM/IDS systems. *Security and Communication Networks*, Hindawi Limited, v. 2020, p. 1–15, 2020. Citado 1 vez na página 15.

MUHAMMAD, Adabi Raihan; SUKARNO, Parman; WARDANA, Aulia Arif. Integrated Security Information and Event Management (SIEM) with Intrusion Detection System (IDS) for Live Analysis based on Machine Learning. *Procedia Computer Science*, Elsevier, v. 217, p. 1406–1415, 2023. Citado 1 vez na página 17.

PELLETIER, Zachariah; ABUALKIBASH, Munther. Evaluating the CIC IDS-2017 dataset using machine learning methods and creating multiple predictive models in the statistical computing language R. *Science*, v. 5, n. 2, p. 187–191, 2020. Citado 4 vezes nas páginas 16–18, 26, 31.

SECURITY, Microsoft. *O que é malware?* 2024. Acessado em: 28 de setembro de 2024. Disponível em: https://www.microsoft.com/pt-br/security/business/security-101/what-is-malware. Citado 2 vezes na página 11.

SHARAFALDIN, Iman; LASHKARI, Arash Habibi; GHORBANI, Ali A et al. Toward generating a new intrusion detection dataset and intrusion traffic characterization. *ICISSp*, v. 1, p. 108–116, 2018. Citado 3 vezes nas páginas 13, 25, 31.

SVIATUN, O V et al. Combating cybercrime: economic and legal aspects. *WSEAS Transactions on Business and Economics*, WSEAS, v. 18, p. 751–762, 2021. Citado 1 vez na página 11.

VENTURES, Cyber Security. *Cybercrime To Cost The World* $10.5 Trillion Annually By 2025$. 2020. Acessado em: 27 de setembro de 2024. Disponível em: https://cybersecurityventures.com/cybercrime-damage-costs-10-trillion-by-2025. Citado 2 vezes na página 12.

YEHOSHUA, Roi. *Random Forest*. 2023. Acessado em: 28 de setembro de 2024. Disponível em: https://medium.com/@roiyeho/random-forests-98892261dc49. Citado 0 vez na página 33.

APÊNDICE A – Dados dos modelos de RNA treinados

Tabela 7 – Métricas do treinamento das séries de modelos de Redes Neurais Artificiais - Parte 1

	Série	Modelo	F-Score	Acurácia	Precisão	Recall
0	0	Full128	0.984600	0.985000	0.998500	0.971000
1	0	Full16	0.996700	0.996800	0.999200	0.994200
2	0	Full32	0.967200	0.968800	0.999700	0.936800
3	0	Full64	0.983200	0.983700	0.997900	0.968900
4	0	Reduced128	0.948100	0.948300	0.936500	0.960000
5	0	Reduced64	0.931800	0.936300	0.984400	0.884500
6	1	Full128	0.988400	0.988800	0.998700	0.978400
7	1	Full16	0.911700	0.918600	0.976400	0.855000
8	1	Full32	0.904700	0.913200	0.982800	0.838100
9	1	Full64	0.984700	0.985200	0.999500	0.970300
10	1	Reduced128	0.925600	0.931100	0.987100	0.871300
11	1	Reduced64	0.965800	0.966600	0.971700	0.960000
12	2	Full128	0.992900	0.993100	0.998000	0.987900
13	2	Full16	0.984500	0.985000	0.999800	0.969700
14	2	Full32	0.967400	0.969000	1.000000	0.936900
15	2	Full64	0.961200	0.963300	0.999800	0.925500
16	2	Reduced128	0.929200	0.934100	0.984400	0.879800
17	2	Reduced64	0.969700	0.970500	0.980200	0.959400
18	3	Full128	0.989400	0.989700	0.998700	0.980300
19	3	Full16	0.990200	0.990500	0.998500	0.982100
20	3	Full32	0.996600	0.996600	0.998700	0.994400
21	3	Full64	0.989600	0.989900	0.998800	0.980600
22	3	Reduced128	0.907000	0.915500	0.988300	0.838100
23	3	Reduced64	0.952200	0.954300	0.981100	0.924900
24	4	Full128	0.984600	0.985100	0.999200	0.970400
25	4	Full16	0.982300	0.982900	0.999400	0.965800
26	4	Full32	0.997400	0.997400	0.998100	0.996600
27	4	Full64	0.984900	0.985400	0.999000	0.971200
28	4	Reduced128	0.803200	0.759200	0.671200	1.000000
29	4	Reduced64	0.930100	0.935700	0.997500	0.871300

Tabela 8 – Métricas do treinamento das séries de modelos de Redes Neurais Artificiais - Parte 2

	Série	Modelo	F-Score	Acurácia	Precisão	Recall
30	5	Full128	0.994900	0.995000	0.998200	0.991700
31	5	Full16	0.982600	0.983100	0.999900	0.965800
32	5	Full32	0.998000	0.998100	0.999100	0.997000
33	5	Full64	0.975900	0.976000	0.965300	0.986700
34	5	Reduced128	0.740300	0.655200	0.587700	1.000000
35	5	Reduced64	0.740500	0.655600	0.588000	1.000000
36	6	Full128	0.984800	0.985300	0.999900	0.970200
37	6	Full16	0.980900	0.981600	0.999800	0.962800
38	6	Full32	0.988300	0.988500	0.994300	0.982300
39	6	Full64	0.992600	0.992800	0.999600	0.985800
40	6	Reduced128	0.925200	0.930300	0.978800	0.877100
41	6	Reduced64	0.961800	0.963200	0.981200	0.943200
42	7	Full128	0.998800	0.998800	0.998100	0.999500
43	7	Full16	0.996100	0.996100	0.993500	0.998700
44	7	Full32	0.992900	0.993000	0.988500	0.997400
45	7	Full64	0.994700	0.994700	0.989600	0.999700
46	7	Reduced128	0.945300	0.944400	0.914900	0.977800
47	7	Reduced64	0.924400	0.929900	0.984400	0.871300
48	8	Full128	0.975300	0.976300	0.999800	0.951900
49	8	Full16	0.977100	0.978000	0.999900	0.955400
50	8	Full32	0.994600	0.994700	0.997000	0.992300
51	8	Full64	0.990400	0.990700	0.999500	0.981500
52	8	Reduced128	0.924400	0.929900	0.984400	0.871300
53	8	Reduced64	0.961500	0.962200	0.963300	0.959800
54	9	Full128	0.990600	0.990800	0.998700	0.982500
55	9	Full16	0.998300	0.998300	0.999100	0.997500
56	9	Full32	0.998500	0.998500	0.998200	0.998700
57	9	Full64	0.986300	0.986700	0.998500	0.974400
58	9	Reduced128	0.928200	0.933800	0.993200	0.871300
59	9	Reduced64	0.906800	0.899900	0.836300	0.990300

APÊNDICE B – Códigos utilizados no treinamento dos modelos do trabalho

Todos os códigos utilizados no desenvolvimento quanto os dados coletados nele, para treinamento e simulação, foram armazenados em três repositórios da ferramenta *Github* e podem ser acessados através dos links:

1. Repositório de treinamento dos modelos e análise de dados(https://github.com/wjunior15/modelos-tcc)
2. Repositório da Simulação (https://github.com/wjunior15/python-soc-tcc)
3. Conjuntos de dados extraídos do ambiente (https://github.com/wjunior15/pcaps-tcc)