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

DHIEGO RAMOS PINTO

APRENDIZADO PROFUNDO APLICADO À ANÁLISE ESTÁTICA DE MALWARES

Rio de Janeiro
2018

INSTITUTO MILITAR DE ENGENHARIA

DHIEGO RAMOS PINTO

APRENDIZADO PROFUNDO APLICADO À ANÁLISE ESTÁTICA DE MALWARES

Dissertação de Mestrado apresentada ao Curso de Mestrado em Sistemas e Computação do Instituto Militar de Engenharia, como requisito parcial para a obtenção do título de Mestre em Ciências em Sistemas e Computação.

Orientador: Prof. Julio Cesar Duarte - D.Sc.

Rio de Janeiro
2018

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

004.69 Pinto, Dhiego Ramos
S586e Aprendizado Profundo Aplicado à Análise Estática de Malwares / Dhiego Ramos Pinto, orientado por Julio Cesar Duarte - Rio de Janeiro: Instituto Militar de Engenharia, 2018.

126p.: il.

Dissertação (mestrado) - Instituto Militar de Engenharia, Rio de Janeiro, 2018.

1. Curso de Sistemas e Computação - teses e dissertações. 1. Inteligência Artificial. 2. Aprendizado de Máquina. 3. Redes Neurais. 4. Aprendizado Profundo. 5. Análise de Malware. I. Duarte, Julio Cesar. II. Título. III. Instituto Militar de Engenharia.

INSTITUTO MILITAR DE ENGENHARIA

DHIEGO RAMOS PINTO

APRENDIZADO PROFUNDO APLICADO À ANÁLISE ESTÁTICA DE MALWARES

Dissertação de Mestrado apresentada ao Curso de Mestrado em Sistemas e Computação do Instituto Militar de Engenharia, como requisito parcial para a obtenção do título de Mestre em Ciências em Sistemas e Computação.

Orientador: Prof. Julio Cesar Duarte - D.Sc.

Aprovada em 20 de Agosto de 2018 pela seguinte Banca Examinadora:

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

Prof. Ronaldo Ribeiro Goldschmidt - D.Sc. do IME

Prof. Eduardo Bezerra da Silva - Ph.D. do CEFET-RJ

Rio de Janeiro
2018

À minha esposa Margareth e filhas Valkíria e Flora, por serem o motivo da minha alegria e constante inspiração. Aos meus irmãos, Caio e Gauer, por sempre me fazerem gargalhar mesmo nos momentos mais tensos. Ao meu pai Ricardo – *In Memorian*– e minha mãe Barbara, por seu amor, exemplo de caráter e todo seu sacrifício para que eu sempre pudesse ter o melhor.

AGRADECIMENTOS

Agradeço a todas as pessoas que me incentivaram e apoiaram esta jornada, de maneira direta ou indireta.

A todo corpo docente e discente do Instituto Militar de Engenharia. Em especial ao meu Professor Orientador Dr. Julio Cesar Duarte, por todas as idéias e conhecimentos passados, por sua paciência, atenção e incentivo.

Aos meus professores da graduação que acreditaram no meu potencial e de alguma forma me encorajaram, Prof. André Sobral, Prof. Adriano Caminha, Prof. Eduardo Bezerra, Prof. Luis Gustavo Hoyer e Prof. Jorge Soares.

Aos meus colegas de turma do IME, por todo o companheirismo e por tornarem o mestrado mais leve nos momentos mais complicados.

Aos meus coordenadores Jacilene Torres e Wilson Luiz, por sua compreensão e todo apoio que me deram no trabalho, sem isso eu não teria conseguido seguir adiante.

Aos amigos Geraldo Souza Júnior e Fabrício Nogueira da Silva, por me incentivarem no retorno à academia e por sempre me emprestarem seus ouvidos. Aos amigos André e Adriana Figueiredo, Wladimyr Proença, Rômulo Dezonne, Gabriel Salles, Gustavo Bunger e Nilson Obermüller por todo o carinho e constante presença.

“Those who can imagine anything, can create the impossible. ”

ALAN TURING

SUMÁRIO

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

RESUMO

Descobrir se um programa suspeito possui código malicioso e categorizá-lo em famílias vem se tornando uma tarefa cada vez mais complexa. Com o crescimento no número de novas gerações de *malware*, a exploração de vulnerabilidades, sejam de hardware ou software, tem sido cada vez maior, gerando prejuízos incontáveis anualmente. Diversas técnicas de aprendizado de máquina tem sido empregadas no contexto da análise de *malware* visando automatizar parte do trabalho do analista, comumente utilizando-se de características pertencentes ao domínio do problema.

O Aprendizado Profundo, uma subárea do Aprendizado de Máquina, despontou na última década como sendo um método de aprendizado de representações, onde hierarquias de representações são criadas nas camadas internas de uma rede neural, necessitando de pouco ou nenhuma engenharia de características para que problemas complexos pudessem ser resolvidos com uma boa performance, e até mesmo, redefinindo o estado da arte em algumas áreas.

Portando, o presente trabalho expõe e experimenta uma abordagem ao problema da classificação de *malware*, utilizando redes profundas com pré-treinamento não supervisionado, aprendendo uma hierarquia de representações, para posteriormente servir como base para classificadores profundos, sem o auxílio de qualquer característica advinda de conhecimento sobre as famílias de *malware*. Os resultados analisados confirmam que a abordagem é bem sucedida nesta tarefa.

ABSTRACT

Deciding if a suspect program has malicious code and categorize it among malware families has become an increasingly complex task. With the growth in the numbers of new malware generations, the exploitation of vulnerabilities, hardware or software based, creates uncountable losses yearly. Several machine learning techniques have been employed in the context of malware analysis, seeking to automate part of an analyst's job, commonly using features belonging to the problem domain.

Deep learning, a subarea of machine learning, emerged in the last decade as a method of representation learning, where representational hierarchies are created in the hidden layers of a neural network, needing little or nothing of engineering feature to achieve a good performance and even more, redefining the state of the art in some areas.

Therefore, the present work exposes and experiments an approach for malware classification, using deep networks with unsupervised pre-training, learning a hierarchy of representations, lately serving as a basis for deep classifiers, without the help of any features that come from the knowledge about the malware's families. The analyzed results confirmed that the approach is well succeeded at this task.

1 INTRODUÇÃO

O ambiente cibernético, nos dias atuais, vem se tornando progressivamente hostil, acarretando no aumento da complexidade dos mecanismos necessários para a sua defesa. Isso se deve em função do aumento gradual de possíveis vetores de propagação de ataques: O uso difundido de aplicativos sociais, o crescente número de dispositivos móveis, dispositivos da "Internet das Coisas" – IoT[1], todos exemplos de tecnologias com vulnerabilidades descobertas e exploradas, sejam elas de hardware, protocolos de rede, aplicativos ou até mesmo a através da exploração de características inerentes da própria tecnologia (JANG-JACCARD; NEPAL, 2014).

Malwares posam como uma das principais ferramentas para execução de atividades ilícitas no ambiente cibernético. Esses programas maliciosos tem como objetivo causar dano ou subverter funções no sistema em que esteja inserido, obviamente, sem o consentimento de seu verdadeiro proprietário (MCGRAW; MORRISETT, 2000). O termo *malware* engloba uma ampla classe de ameaças, como vírus, *trojans*, *bots*, *ransonwares*, dentre outros, porém, não limitam-se a encaixarem em apenas uma categoria: comumente possuem comportamento heterogêneo e são programados com sofisticadas técnicas de proteção e ofuscação, permitindo-lhes evadir de mecanismos de detecção e classificação, tornando o trabalho de especialistas em segurança complexo e moroso (JANG-JACCARD; NEPAL, 2014; DAMSHENAS et al., 2013).

A morosidade no trabalho de análise dos *malwares* leva a necessidade de ferramentas automatizadas e inteligentes, capazes de aprender baseando-se no conhecimento de especialistas ou de conhecimento útil retirado intrinsecamente dos dados, ou seja, diretamente do *malware*.

Nesta última categoria, podemos destacar o emprego da Inteligência Artificial, particularmente do Aprendizado de máquina, para o processo de automatização da análise do *malware*, tanto estática, onde analisa-se o arquivo sem executá-lo, quanto a dinâmica, onde utiliza-se padrões do arquivo executável durante sua execução. Abordagens utilizando redes neurais de aprendizado profundo vem ganhando notoriedade na resolução de diversos problemas nas mais diversas áreas, incluindo a defesa cibernética.

[1]IoT, do inglês, Internet of Things

1.1 MOTIVAÇÃO

Nos últimos anos, pudemos acompanhar grandes avanços em Aprendizado de Máquina, especificamente nos modelos de Aprendizado Profundo[2], onde com o emprego desses modelos, pudemos alcançar o estado da arte em diversos problemas complexos da computação (LECUN et al., 2015).

O aprendizado da representação dos dados é a grande vantagem das abordagens envolvendo Aprendizado Profundo, pois permitem que esses modelos aprendam a realizar a extração de características dos dados automaticamente, independendo da engenharia de características. Podemos afirmar que esses modelos aprendem uma Hierarquia de Conceitos, em que a cada camada superior da rede neural representa um conceito mais abstrato, baseado em um conceito mais simples aprendido por uma camada anterior (GOODFELLOW et al., 2016; BENGIO et al., 2013).

Tradicionalmente, uma das etapas custosas no desenvolvimento de sistemas de aprendizado de máquina está na construção da representação dos dados. Geralmente essas características são selecionadas e tratadas por especialistas no domínio, devendo ser suficientes para que os modelos treinados pudessem alcançar uma boa performance. Com o desenvolvimento de abordagens utilizando o aprendizado profundo, podemos observar que essas técnicas necessitam de pouca ou nenhuma engenharia de características para obterem boa performance, já que o próprio modelo é o encarregado de realizar a seleção das melhores características.

Apenas pelos resultados alcançados por essas arquiteturas de redes neurais em diversas áreas e problemas de grande complexidade já seria motivo suficiente para realizar uma investigação aprofundada sobre sua aplicação no contexto da *Análise de Malware*, uma vez que, essas ameaças continuam sendo de difícil detecção e classificação. O *malware* já vem embarcado com diversas técnicas de evasão, muitas vezes o fazendo ser confundido com um programa benigno, complicando sua análise.

Além disso, o código malicioso é uma arma amplamente utilizada em crimes cibernéticos. De acordo com um relatório da Verizon, mais de 50% de incidentes em que ocorreram vazamentos de dados empregou algum tipo de malware em algum ponto da cadeia de ataque – Seja o malware introduzido ao início do ataque ou durante a sua evolução (VERIZON, 2016). Vimos também, nos últimos anos, um aumento de ataques com *ransomware*, o crescimento vertiginoso do uso de dispositivos da Internet das Coisas como vetor de ataques, descobrimento de grandes *botnets* e um aumento de 54% no número

[2]Do inglês, Deep Learning

de variantes de malware para dispositivos *mobile*, o aumento de infecções por *coinminers*, *trojans* que utilizam os recursos das máquinas que infectaram para mineração de criptomoedas (SYMANTEC, 2018). No contexto atual, o malware está cada vez mais sofisticado, e as previsões para o futuro continuam alarmantes.

Sendo assim, acreditamos que o emprego de técnicas avançadas de aprendizado de máquina, como o Aprendizado Profundo, podem auxiliar no combate a esse tipo de ameaça.

1.2 CARACTERIZAÇÃO DO PROBLEMA

Malwares são uma classe de ataques de difícil identificação. O trabalho do analista de segurança para a detecção e identificação de um arquivo como malicioso é demorado e envolve o uso de diversas ferramentas, como antivírus, *debuggers*, *disassemblers*, e comumente a montagem de ambientes preparados para a observação do arquivo durante a sua execução. Os programadores de *malware* dificultam ainda mais o trabalho, incrementando seu código com métodos de evasão, criando novas gerações de *malware* que exploram novas falhas de segurança. Do início de uma análise até a confirmação do *malware* pode ser demorado. Apesar de existirem diversas ferramentas que automatizam a tarefa de análise do *malware*, podemos observar que ainda existe uma grande dependência do fator humano em todo o processo. A quantidade de arquivos a serem avaliados por um analista, mesmo depois do emprego das ferramentas automatizadas, ainda é muito grande, tornando o processo de atualização de sistemas de segurança ineficaz diante da velocidade da evolução dessas ameaças. Portanto, é imperativo o desenvolvimento de novos métodos automatizados, inteligentes e eficazes na detecção e classificação de *malwares* que sejam resilientes as possíveis evoluções que seu código possa sofrer e que dependam minimamente de um especialista, diminuindo assim a sua carga de trabalho.

1.3 OBJETIVOS

Este trabalho tem como objetivo investigar a viabilidade da aplicação de arquiteturas de redes neurais profundas na análise estática de *malware*, mais precisamente para classificação dessas ameaças, utilizando-se apenas de seu código-fonte desmontado, desprezando qualquer característica ou informação advinda de conhecimento prévio sobre cada família de *malware* pertencente as amostras utilizadas no estudo.

Como objetivos específicos podemos destacar:

- A investigação do uso de contagens de OpCodes e operandos extraídos de *malware* desmontado, apresentada no formato de "Bag of OpCodes", com variações em n-gramas, ponderados por sua frequência, como boas características para a Classificação de Malware

- Investigar o uso de redes neurais Autocodificadoras Profundas como descobridora de boas representações de dados advindas de código desmontado de *malware* e sua eficiência como pré-inicializadora de rede supervisionada Classificação de *Malware*

- Realizar uma análise de sensibilidade das redes neurais experimentadas, verificando seu desempenho e como reagem a alterações em seus parâmetros

1.4 JUSTIFICATIVA

É difícil de se contabilizar o custo que os ataques cibernéticos provocam na economia mundial. De acordo com o relatório *"Net Losses: Estimating the Global Cost of Cybercrime"* (MCAFEE, 2014) estima-se que o crime cibernético custe anualmente entre 375 e 575 bilhões de dólares em perdas. Em um relatório mais recente à Casa Branca (OF ECONOMIC ADVISERS, 2018), o custo de atividades cibernéticas ficou estimado de 57 a 109 bilhões de dólares em prejuízos causados por ameaças cibernéticas somente nos Estados Unidos.

Essas ameaças colocam em risco a integridade de infraestruturas críticas para um país, podendo afetar o funcionamento de sistemas e órgãos ligados aos setores de Defesa, Energia, Transporte, Telecomunicações, Finanças, dentre outros. Devido a criticidade desses setores, a Defesa Cibernética vêm se destacando como função de Estado em diversos países. No Brasil, o setor Cibernético é considerado estratégico para a segurança nacional, conforme definido pela Estratégia Nacional de Defesa, ficando a cargo do Exército Brasileiro a garantia de sua segurança (PRESIDÊNCIA DA REPÚBLICA, 2010; MINISTÉRIO DA DEFESA , 2012).

A preocupação com a segurança cibernética não é apenas do setor público e se estende tanto para o mercado privado como para aos usuários finais de tecnologia. De acordo com o *Data Breach Investigations Report*, 90% das violações de dados identificadas como espionagem cibernética ocorridas em 2015 tinham como objetivo o roubo de segredos comerciais e de informações de patentes. Em 2016, 90% dos incidentes com espionagem cibernética tinham envolvimento de *sotfware* malicioso. O mesmo relatório afirma que 89% de todas as violações de dados tem como motivação o ganho financeiro ou a espionagem

(VERISON, 2016).

Com o passar do tempo, a situação se torna mais complexa. O mundo comprou cerca de 1.4 bilhões de novos *smartphones* em 2015, 10% a mais do que no ano anterior. Hoje temos 248 milhões de *tablets* e estima-se que esse número esteja em torno de 269 milhões em 2019. Teremos 780 milhões de dispositivos vestíveis, como relógios inteligentes em 2018. Dispositivos IoT, como controladores de luzes, trancas de portas, câmeras de segurança, TVs inteligentes, sensores, entre outros, serão 200 bilhões em 2020. Estima-se que em 2020 haverá 220 milhões de automóveis conectados a rede (MCAFEE, 2015; SYMANTEC, 2016).

Independente do cenário futuro, já existe hoje uma grande demanda por ferramentas inteligentes que auxiliem o trabalho de detecção e classificação de *malwares*. O Aprendizado Profundo despontou nos últimos anos com resultados expressivos em problemas de grande complexidade, ultrapassando o resultado de diversos algoritmos de aprendizado de máquina clássicos nas mais diversas tarefas (LECUN et al., 2015). Acreditamos que a abordagem desenvolvida nesta pesquisa possa contribuir com a melhoria dos métodos de detecção e classificação de malware, auxiliando o exército em sua missão.

1.5 METODOLOGIA

Para obtenção do conhecimento necessário e validação do problema escolhido, foi realizada uma extensa pesquisa bibliográfica focada nos principais assuntos abordados pelo estudo: Aprendizado de Máquina, mais precisamente o Aprendizado Profundo e a Análise de *Malware*. A pesquisa foi conduzida de maneira experimental, procurando assim medir e validar a aplicabilidade da abordagem proposta em relação ao problema a ser resolvido. A análise dos resultados experimentais foi feita de maneira qualitativa e quantitativa.

1.6 ESTRUTURA DO TEXTO

Esta dissertação está organizada da seguinte maneira: O capítulo 2 discorre sobre o aprendizado de máquina, enfatizando aspectos relacionados ao Aprendizado Profundo [3] e suas arquiteturas mais conhecidas. O capítulo 3 introduz o conceito de *malware*, apresentando suas características, técnicas de evasão e conceitos relativos a análise de malware. O capítulo 4 descreve trabalhos científicos que possuem alguma relação com este estudo. O capítulo 5 descreve a abordagem criada durante este estudo. O capítulo 6 especifica a configuração dos experimentos executados e discute-se os resultados de suas

[3] do inglês, *Deep Learning*

execuções. Por fim, o capítulo 7 apresenta a conclusão deste trabalho, considerações finais e sugestões para trabalhos futuros.

2 APRENDIZADO DE MÁQUINA

O Aprendizado de Máquina é um campo de estudo que surgiu da Inteligência Artificial e está preocupado com a criação de algoritmos que consigam aprender diretamente da observação de dados, considerando seus erros e acertos, em um processo de aprendizado automático, capacitando-o a realizar a mesma tarefa aprendida em dados até então não observados. Uma das primeiras definições sobre o assunto foi criada por Arthur L Samuel (1959) que descreveu o Aprendizado de Máquina como sendo o campo de estudo que visa habilitar computadores a aprenderem sem terem sido explicitamente programados. Mitchell (1997) definiu o Aprendizado de Máquina como sendo o campo de estudo que se preocupa com a questão de como construir programas de computador que automaticamente melhoram com a experiência, ou seja, diz-se que:

> *"Um programa de computador aprende de uma experiência E em relação a uma classe de tarefas T e medida de performance P, se sua performance nas tarefas em T, medidas por P, melhoram com experiência E".*

Adotando essa definição, poderíamos utilizar como exemplo um classificador de dígitos manuscritos. Neste contexto, teríamos T como a tarefa de reconhecer dígitos manuscritos a partir de imagens; P seria a medida de performance, como por exemplo, o percentual de amostras corretamente classificadas; e por fim, E, a experiência adquirida através do treinamento em um conjunto de dados rotulado de imagens de dígitos manuscritos, como por exemplo, o MNIST[4] (MITCHELL, 1997).

O objetivo fundamental de todo algoritmo de Aprendizado de Máquina é a **generalização**, o que significa afirmar que ele deve ser capaz de generalizar para além das amostras que foram observadas durante seu treinamento (DOMINGOS, 2012). Segundo Monard e Baranauskas (2003), a indução é uma forma de inferência lógica que permite obter conclusões baseadas em um conjunto particular de exemplos. Portanto, por meio da inferência indutiva aplicada aos exemplos fornecidos, hipóteses – que podem ou não corresponder a verdade – são geradas. Quando essas hipóteses conseguem ser confirmadas satisfatoriamente com exemplos que nunca foram observados, podemos então dizer que ocorreu o aprendizado.

[4]The MNIST Database of handwritten digits; http://yann.lecun.com/exdb/mnist/

Uma série de fatores impactam diretamente no resultado de um modelo, isto é, sua capacidade de generalização, que devem ser observados para que ele seja efetivo. Segundo Goodfellow et al. (2016), dois fatores são determinantes para avaliar o bom desempenho de um algoritmo de aprendizado de máquina. O primeiro, que ocorre durante a fase de treinamento, é a capacidade de fazer com que o erro seja baixo. O outro fator é a capacidade de fazer a diferença entre o erro medido na fase de treinamento e o erro medido na fase de testes do modelo seja pequena. Esses dois fatores correspondem a dois importantes desafios do Aprendizado de Máquina: O ***underfitting*** – **subajustamento** – **e o** ***overfitting*** – **superajustamento** (Figura 2.1).

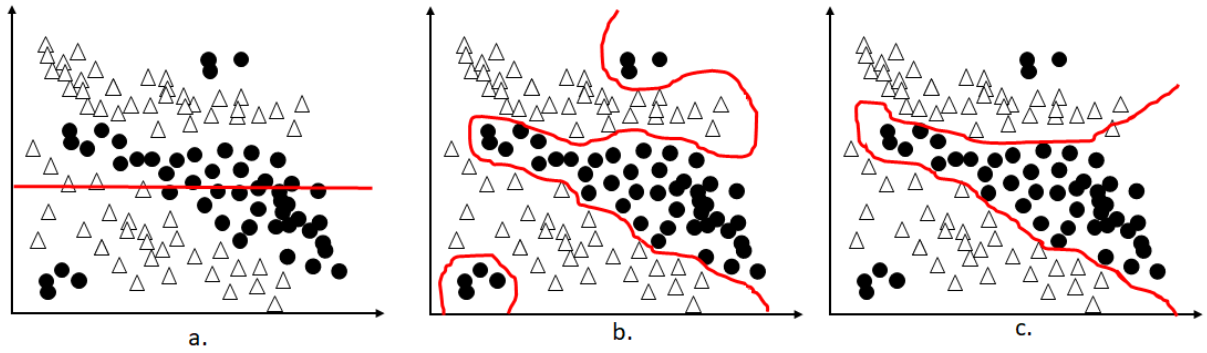

FIG. 2.1: Exemplos de **a.** subajustamento, **b.** superajustamento e **c.** ajuste equilibrado.

Quando uma hipótese induzida obtém uma melhora de desempenho muito pequena no conjunto de treinamento, isto é, a hipótese se ajusta pouco ao conjunto de treinamento, diz-se que ocorreu um **subajustamento**. No entanto, quando uma hipótese se ajusta em excesso aos dados do conjunto de treinamento, considera-se que houve um **superajustamento** do modelo. É possível evidenciar de forma simples se um modelo sofre de superajustamento: se ele possuir uma boa performance no conjunto de treinamento porém ao ser apresentado a um conjunto de dados não observado anteriormente, denominado conjunto de testes, este possuir uma performance muito baixa acarretando em um grande hiato entre o erro medido no conjunto de treinamento e o medido no conjunto de testes (GOODFELLOW et al., 2016; MURPHY, 2012; MONARD; BARANAUSKAS, 2003) .

Para entender melhor o problema, podemos decompor o erro de generalização em dois termos: viés e variância. O erro de viés representa a tendência que o modelo tem em aprender sistematicamente de maneira incorreta. O erro de variância representa a tendência em aprender algo randomicamente, independente do sinal verdadeiro. O superajustamento de um modelo ocorre quando se detecta uma alta variância e um baixo viés.

Já o subajustamento ocorre em modelos com alto viés e baixa variância. Ou seja, se o modelo ignorar demais os dados teremos o subajustamento. De outra forma, se os considerar demais, teremos um superajustamento. Quando criamos um modelo, devemos enfrentar este dilema [5] (Figura 2.2), buscando sempre o "melhor" ajuste, isto é, o equilíbrio entre esses dois termos (DOMINGOS, 2012; FACELI, 2011).

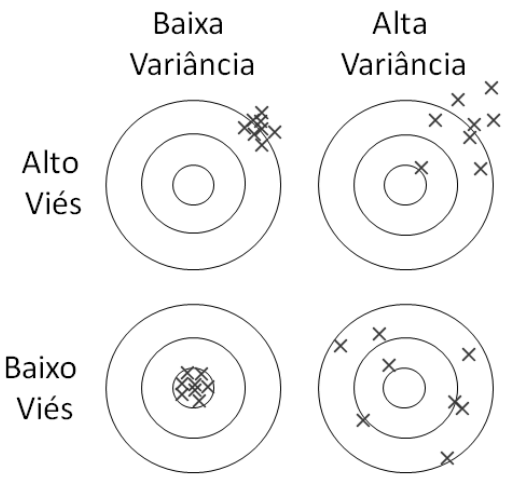

FIG. 2.2: Exemplo do efeito do viés e variância aplicado ao contexto do lançamento de dardos. Adaptada de Domingos (2012)

2.1 PARADIGMAS DE APRENDIZADO

Podemos então classificar o aprendizado indutivo em **Aprendizado Supervisionado** e **Aprendizado Não-Supervisionado**.

No **Aprendizado Supervisionado ou Aprendizado Preditivo**, o objetivo é aprender um mapeamento originando-se de entradas x para saídas y, utilizando um conjunto de tuplas $\{xi, yi\}$, onde y_i é o rótulo correto da classe a qual pertence os valores de entrada x_i, portanto, o conjunto de treinamento pode ser definido como $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$, sendo N o número de exemplos do conjunto. O vetor de características x_i, também conhecido como vetor de atributos ou vetor de covariadas, é o vetor de entrada D-dimensional que pode ser composto de números simples ou estruturas mais complexas, como por exemplo, uma imagem, uma sentença de texto, um grafo ou uma série temporal. A saída poderia ser igualmente em qualquer formato, mas assume-se usualmente que y_i é uma

[5] Este dilema é conhecido como Dilema Viés-Variância – do inglês, *Bias-Variance Dilemma* – ou o Custo-Benefício Viés-Variância – do inglês, *Bias-Variance tradeoff*.

variável categórica, oriunda de algum conjunto finito – $y_i \in \{1, \ldots, C\}$ – ou que seja um escalar de valor real. A saída y_i será definida pelo tipo de problema a ser resolvido. Se o objetivo a ser cumprido for categorizar a entrada x_i, ou seja, y_i é uma variável categórica, diz-se que o problema é um **Problema de Classificação ou Reconhecimento de Padrões**. Todavia, se o objetivo for encontrar um valor real para y_i, o problema é conhecido como **Problema de Regressão** (MURPHY, 2012).

Murphy (2012) formaliza o problema da **classificação** como uma aproximação de função. Assume-se que $y = f(x)$ para alguma função f desconhecida. O objetivo do aprendizado é estimar a função f dado um conjunto de treinamento e então ser capaz de fazer predições usando $\hat{y} = \hat{f}(x)$. Conforme apresentado, o objetivo principal do classificador será o de fazer predições em entradas não observadas anteriormente. A nomenclatura do problema também muda conforme a quantidade de classes. Sendo C o número de classes possível para a saída do classificador. Se $C = 2$ dizemos que o problema é uma **classificação binária** – usualmente $y \in \{0, 1\}$. Contudo, se $C > 2$, chamamos de **classificação multi-classe**. Outrossim, ainda existem problemas em que as classes não são mutuamente exclusivas. Neste caso, chamamos o problema de **classificação multi-rótulo**. Podemos citar como alguns exemplos de classificação: classificação de documentos, como por exemplo um filtro anti-*spam*; classificação de imagens, detecção e reconhecimento facial, classificação de malwares em suas famílias, dentre outros.

Já a **regressão** é similar a classificação exceto pela sua variável de saída, que é contínua. O objetivo do classificador é retornar uma função $f : \mathbb{R}^n \to \mathbb{R}$. Como exemplo de tarefas de regressão, temos: Predição do preço de uma ação para amanhã, dado o estado atual do mercado; predizer a idade de um expectador visualizando um filme online, predizer o preço de venda de uma casa dada sua localização e outros atributos, entre outros (GOODFELLOW et al., 2016; MURPHY, 2012).

Outro método de aprendizado é conhecido como **Aprendizado Não-Supervisionado ou Aprendizado Descritivo**. Nesta abordagem, apenas possuímos em nosso conjunto de dados amostras não rotuladas, ou seja, $\mathcal{D} = \{x_i\}_{i=1}^{N}$. O objetivo desta categoria de aprendizagem é a **Descoberta de Conhecimento**, isto é, encontrar padrões diretamente dos dados de entrada. Este tipo de aprendizado é o mais próximo do aprendizado humano ou animal. Podemos considerar que dados rotulados além de serem custosos de se produzir, agregam pouca informação, geralmente insuficientes para estimar parâmetros de modelos mais complexos (MURPHY, 2012). Podemos citar como problemas em que o aprendizado não-supervisionado é comumente aplicado a **clusterização** e a **redução de dimensionalidade**.

Um exemplo clássico de aprendizado não-supervisionado é o problema de **clusterização**. Na clusterização, o algoritmo analisa um conjunto de dados e determina se alguns deles podem formar agrupamentos. Uma análise posterior é necessária para determinar o que representa cada agrupamento dentro do contexto do problema analisado (MONARD; BARANAUSKAS, 2003). De acordo com Murphy (2012), o problema pode ser formalizado da seguinte maneira: Seja K a quantidade de agrupamentos ou clusters. O objetivo é estimar a distribuição sobre o número de clusters $p(K|\mathcal{D})$; com isso, saberemos se existem subpopulações entre os dados observados. Por simplicidade, aproxima-se a distribuição $p(K|\mathcal{D})$ por sua moda, $K^* = argmax_K p(K|\mathcal{D})$. Como objetivo secundário, procura-se estimar a qual cluster pertence cada amostra. Seja $z_i \in \{1, \ldots, K\}$ a representação do cluster ao qual o dado i foi atribuído, neste caso, z_i é um exemplo de um dado que não foi observado no conjunto de treinamento. O algoritmo poderá inferir a qual cluster cada amostra pertence calculando $z_i^* = argmax_k p(z_i = k|X_i, \mathcal{D})$.

Um outro exemplo de aplicação do aprendizado não-supervisionado é a **redução de dimensionalidade**. Quando temos que lidar com dados de dimensões elevadas, geralmente é útil realizar uma projeção dos dados para um subespaço dimensional inferior que captura os dados latentes, ou seja, as características essenciais advindas dos dados, filtrando características que não são importantes. De acordo com Murphy (2012), a motivação para a aplicação da redução de dimensionalidade é que, apesar dos dados estarem em uma dimensão superior, devem existir apenas poucos graus de variabilidade, correspondendo a fatores latentes. Representações projetadas em espaço dimensional inferior podem melhorar a acurácia em predições quando utilizadas como entradas em outros modelos estatísticos. Também são úteis para visualização 2D de dados de dimensionalidade alta e por permitirem buscas por vizinhos próximos mais rapidamente. Podemos citar como abordagens comuns para a redução de dimensionalidade o uso de PCA – *principal components analysis*(MURPHY, 2012) ou através da utilização de redes neurais autocodificadoras profundas (HINTON; SALAKHUTDINOV, 2006).

Contudo, esta classificação de aprendizado em supervisionado e não supervisionado pode ser flexível. Como a construção de conjunto de dados rotulados é custosa, temos então o caso em que um modelo aprende mesclando o aprendizado supervisionado e o não supervisionado, chamado de **Aprendizado Semi Supervisionado**. Assim sendo, tanto exemplos rotulados como não rotulados são utilizados no treinamento do modelo. Desta maneira, o algoritmo procura agrupar as amostras não rotuladas baseando-se, por exemplo, em alguma medida de similaridade.

Podemos dividir o Aprendizado Semi supervisionado em duas áreas (SADARAN-

GANI; JIVANI, 2016): Classificação Semi Supervisionada, onde um classificador é treinado tanto com dados rotulados como não rotulados, com o objetivo de obter um modelo mais acurado, e a Clusterização ou Agrupamento Semi Supervisionado, onde as amostras rotuladas ajudam na clusterização de dados não rotulados. Dentre os algoritmos Semi Supervisionados mais utilizados estão o **Auto-treinamento** (do inglês, *Self-training*), onde um classificador é treinado nos dados rotulados e logo em seguida é usado para classificar os dados não rotulados. Dados não rotulados que foram previstos com um alto grau de confiança passam então a fazerem parte do conjunto de treinamento rotulado. Novamente então o classificador é treinado no conjunto de treinamento e o ciclo se repete. Já o **Co-treinamento**(do inglês, *Co-training*), dependem de duas visões dissimilares do conjunto de características, geralmente independentes, porém fornecendo informações complementares sobre uma instância de classe. O aprendizado se inicia com cada classificador sendo treinado em uma das visões dos dados rotulados. Posteriormente, eles são usados para classificarem os dados não rotulados: as amostras previstas com um maior grau de confiança pelos classificadores são então rotuladas. Outro é o **Aprendizado Multi Visão**, que é uma generalização do co-treinamento, que envolve o emprego de vários classificadores que são treinados no mesmo conjunto de dados rotulado e são posteriormente utilizados para predizerem dados de um conjunto não rotulado (HAJIGHORBANI et al., 2016; PRAKASH; NITHYA, 2014).

2.2 REDES NEURAIS ARTIFICIAIS

Uma rede neural artificial pode ser definida como um conjunto de neurônios artificiais conectados, onde um sinal é processado e pode ser suprimido ou reforçado através de uma função de ativação. Neste contexto, neurônios são simples unidades de processamento, que recebem sinais de entrada e a partir da função de ativação e produzem um valor real na sua saída (SCHMIDHUBER, 2015). São modelos inspirados nas redes neurais biológicas: Os neurônios são os blocos de construção de uma rede neural artificial, são simples dispositivos computacionais altamente conectados são simplificações da célula biológica, o mecanismo de funcionamento das sinapses pode ser comparado a ativação de neurônios em uma rede neural artificial. Observamos também que ambas as redes – biológica e artificial – possuem como característica uma grande capacidade de processamento paralelo (DEMUTH et al., 2014).

Um dos trabalhos considerados basilares na modelagem do comportamento de um neurônio natural e consequentemente suas conexões foi desenvolvido por McCulloch e

Pitts (1943). Este modelo de neurônio era binário e ativado por uma função com um limiar pré-fixado – chamado de θ . Este neurônio recebia valores de entrada de sinapses excitatórias, todas com pesos idênticos. Qualquer entrada inibitória tinha poder de veto sobre as entradas excitatórias. Outro detalhe importante é que o neurônio funcionava de maneira síncrona, ou seja, a cada intervalo de tempo eram atualizados pela soma ponderada de suas entradas excitatórias, produzindo um valor de saída igual a um, se e somente se, o valor da soma fosse maior ou igual ao valor de θ, considerando que o neurônio não tenha recebido nenhuma entrada inibitória. Em caso contrário, o neurônio produziria um valor de saída igual a zero. Com este trabalho, os autores conseguiram desenvolver o primeiro modelo de um neurônio artificial e conseguiram demonstrar que ao utilizarem conjuntos desses neurônios conectados em determinados padrões conseguiam codificar qualquer proposição lógica.

Posteriormente, Rosenblatt (1958), baseado no modelo de neurônio de McCulloch e Pitts (1943), propôs o *Perceptron*, uma rede simples composta de apenas uma unidade de cálculo, possuindo pesos sinápticos e um termo de viés ou *bias*. Rosenblatt também criou o algoritmo de aprendizado do Perceptron, o primeiro utilizando aprendizado supervisionado, e também foi o responsável por provar sua convergência, através do Teorema de Convergência do Perceptron, que basicamente afirmava que se os padrões – vetores – utilizados no treinamento do Perceptron são extraídos de duas classes linearmente separáveis, então o algoritmo converge e posiciona uma fronteira de decisão entre essas duas classes (HAYKIN, 2004). Podemos então afirmar que uma rede Perceptron constituída apenas de um neurônio está limitada a realizar tarefas de classificação binária.

2.3 DO NEURÔNIO ARTIFICIAL À REDE DE MÚLTIPLAS CAMADAS

Baseando-se em Demuth et al. (2014), detalharemos os componentes de um modelo de neurônio artificial com apenas um valor de entrada e o expandiremos até um neurônio de múltiplas entradas. Posteriormente, detalharemos uma rede neural, da arquitetura mais simples, apenas uma camada, até uma arquitetura de rede com múltiplas camadas.

2.3.1 O NEURÔNIO ARTIFICIAL

A fórmula da saída de um **neurônio simples com apenas um valor de entrada** (Figura 2.3) pode ser definida por

$$y = f(wx + b), \tag{2.1}$$

sendo x um valor escalar de entrada do neurônio; w, o peso do neurônio, também um escalar, formando wx, o termo a ser enviado a função de soma. A outra entrada, geralmente "1" é multiplicada por b, que é o termo conhecido como *bias* ou viés, que também é enviado para a função de soma. A função de soma produz uma saída n que passa então por uma função de ativação f, produzindo a saída do neurônio, o escalar y. Exemplificando, se tivermos $w = 2$, $x = 1,5$ e $b = -1,3$, então teríamos

$$a = f(3(1,5) - 1,5) = f(1,7)$$

Podemos observar que o valor de y dependerá da função de ativação – f – escolhida. O *bias* ou viés funciona também como um peso, com a constante "1" sendo a única diferença a se observar. Observa-se também que w e b são parâmetros do neurônio que são ajustáveis, ajuste esse que é realizado comumente por alguma regra de aprendizado.

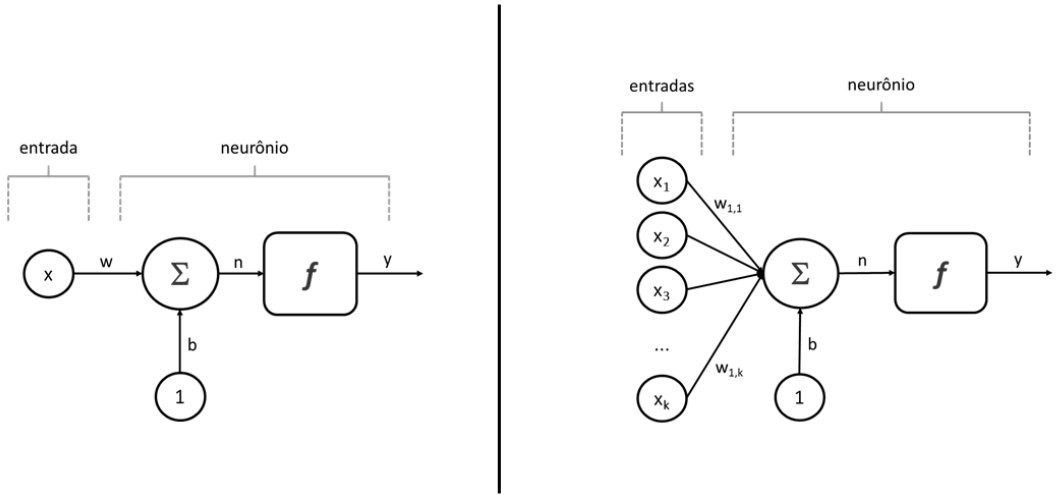

FIG. 2.3: Ao lado **esquerdo**, podemos observar uma representação gráfica de um neurônio simples com apenas uma entrada escalar. No canto **direito**, vemos a representação gráfica de um neurônio com K entradas. Adaptada de Demuth et al. (2014)

Fazendo um paralelo entre este modelo matemático de neurônio e o neurônio biológico, w representa a força de uma sinapse, a função de soma em conjunto com a função de ativação representa o corpo da célula, e y, a saída do neurônio, representa o axônio.

Para um **neurônio com K entradas** (Figura 2.3), podemos observar cada entrada individual $x_1, x_2, x_3, \ldots, x_K$ sendo ponderada pelos pesos $w_{1,1}, w_{1,2}, w_{1,3}, \ldots, w_{1,K}$ oriundos de uma matriz de pesos sinápticos W. Este neurônio possui um *viés* b que, somado às entradas ponderadas, resulta em

$$n = w_{1,1}x_1 + w_{1,2}x_2, w_{1,3}x_3 + \cdots + w_{1,K}x_K + b$$

que, sendo escrito em notação matricial, torna-se:

$$n = Wx + b,$$

com a saída do neurônio resultando em

$$y = f(Wx + b), \tag{2.2}$$

e o valor resultante y, dependente da função de ativação escolhida.

2.3.2 ARQUITETURA EM CAMADAS

Geralmente uma rede neural envolve o uso de um número maior de neurônios interconectados. Esses neurônios operam em paralelo e são dispostos no que chamamos de camadas.

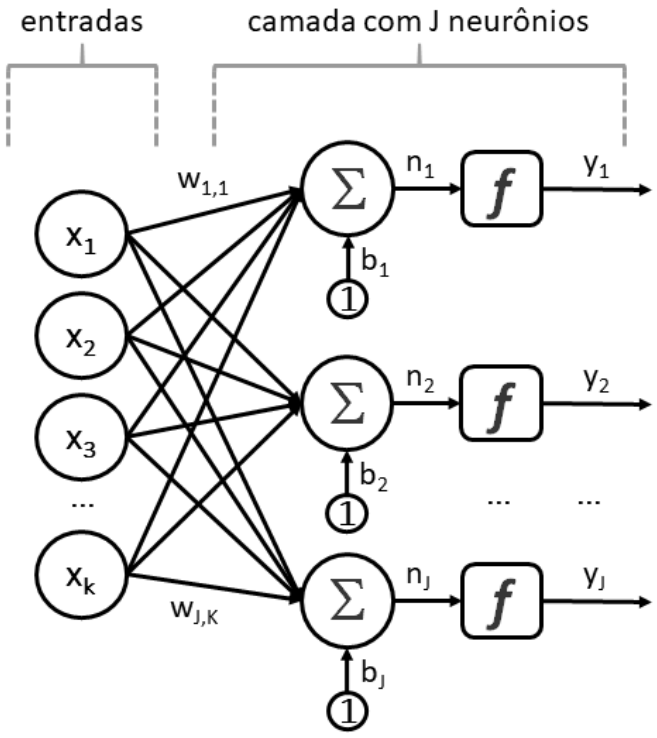

FIG. 2.4: Arquitetura de rede neural de camada única com J neurônios. Adaptada de Demuth et al. (2014)

Analisando uma **rede neural de camada única** (Figura 2.4), podemos observar que os elementos de entrada da rede formam um vetor x que está conectado a cada um dos neurônios da camada através de uma matriz de pesos W:

$$W = \begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,K} \\ w_{2,1} & w_{2,2} & \dots & w_{2,K} \\ \vdots & \vdots & & \vdots \\ w_{J,1} & w_{J,2} & \dots & w_{J,K} \end{bmatrix} \tag{2.3}$$

Cabe ressaltar que, em relação a matriz de pesos, o índice de linha da matriz indica o neurônio de destino associado ao peso e o índice da coluna da matriz indicam a origem da entrada associado ao peso – por exemplo, $w_{2,2}$ representa a conexão de x_2 com o segundo neurônio da camada. Cada um dos neurônios possui um termo de viés ou *bias* b_i, uma função de soma, a função de ativação f e a saída y_i, o que para a camada significa dizer que temos um vetor de vieses b e um vetor de saídas y. Portanto, para calcularmos a saída da rede camada única, temos:

$$y = f(Wx + b) \tag{2.4}$$

Examinemos então uma **rede neural com múltiplas camadas**. Neste caso, cada camada da rede terá sua própria matriz de pesos W, seu próprio vetor de vieses b e um vetor com as saídas calculadas y.

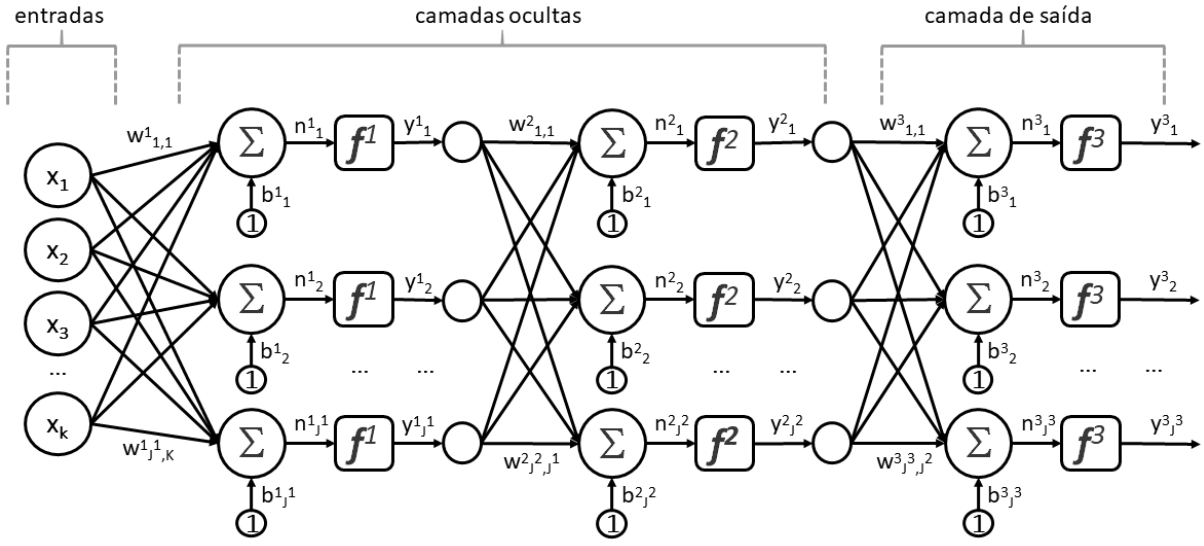

FIG. 2.5: Arquitetura de rede neural com múltiplas camadas – uma camada de entrada, duas camadas ocultas e uma camada de saída. Adaptada de Demuth et al. (2014)

No caso de mais de uma camada, devemos destacar que as entradas de uma determinada camada U são as saídas da camada $U-1$. As camadas situadas entre a camada de entrada e a de saída da rede são chamados de camadas ocultas. Como exposto na fi-

gura 2.5, existem K entradas, J^1 neurônios na primeira camada, J^2 neurônios na segunda camada, e assim por diante. Assim sendo, podemos calcular a saída da rede como uma composição das funções de todas as camadas:

$$y^1 = f^1(W^1 x + b^1) \tag{2.5}$$

$$y^2 = f^2(W^2 y^1 + b^2) \tag{2.6}$$

$$y^3 = f^3(W^3 y^2 + b^3) = f^3(W^3 f^2(W^2 f^1(W^1 x + b^1) + b^2) + b^3) \tag{2.7}$$

2.3.3 TREINAMENTO POR *BACK-PROPAGATION*

Segundo FACELI (2011), o algoritmo *back-propagation* é um algoritmo de treinamento que tem seu funcionamento constituído em duas fases: *forward* ou "para frente" e *backward* ou "para trás". Inicialmente, o *back-propagation* necessita que os pesos sinápticos da rede tenham sido inicializados – comumente são adotados valores randômicos não uniformes para os pesos. Sendo assim, podemos então decompor o algoritmo em 4 etapas menores: ***forward propagation***, ***back-propagation* para a camada de saída**, ***back-propagation* para camadas intermediárias** e por fim a **atualização dos pesos** da rede. Detalharemos, a seguir, cada etapa individualmente.

Inicialmente os valores de entrada da rede são passados para cada neurônio da camada intermediária, ponderando-se pelo valor de seus pesos associados as conexões de entrada. Cada neurônio desta camada aplica então a função de ativação ou transferência a sua entrada total, produzindo assim um valor de saída que será utilizado como valor de entrada na próxima camada. Isto se repete em camadas intermediárias até que se atinja a camada de saída da rede (etapa ***forward propagation***).

Ao atingir a camada de saída da rede, calcula-se o erro total da rede (etapa ***back-propagation* para a camada de saída**). Este erro é calculado à partir da discrepância entre os valores produzidos pelos neurônios de saída e os valores desejados para cada um deles. Calculado o valor de erro para cada neurônio de saída, o algoritmo calcula o erro das camadas intermediárias iterativamente até a primeira camada intemediária da rede (etapa ***back-propagation* para camadas intermediárias**). A equação de cálculo do erro depende da camada que a iteração do algoritmo se encontra. Na camada de saída, o algoritmo já conhece os valores de erro para cada neurônio. Já nas camadas intermediárias, necessita-se que uma estimativa seja calculada, por isso o *back-propagation* estima os valores de erro dos neurônios de tais camadas baseado na soma dos erros de todos os neurônios conectados da camada posterior, ponderado pelo valor de seus pesos

sinápticos.

Dessa forma, realiza-se o cálculo das derivadas parciais definindo assim o ajuste que deverá ocorrer nos pesos (etapa de **atualização dos pesos**), utilizando o gradiente descendente da função de ativação de cada neurônio. Assim, consegue-se calcular a influência que cada peso da rede teve – erro parcial – em relação ao erro total medido em sua saída, dada uma classificação de um determinado objeto de entrada. Se o valor calculado da derivada for negativo, significa que o peso em questão contribuiu para que o valor de saída seja mais próximo do desejado, então este peso será aumentado proporcionalmente. Caso contrário, significa que o peso em questão contribuiu para um aumento no erro total da rede, então este peso será diminuído proporcionalmente.

Quanto ao critério de parada, usualmente executa-se o algoritmo até que um critério definido seja atingido. Esse critério pode ser, por exemplo, o número de épocas executando no conjunto de treinamento. Geralmente, para que se evite o superajustamento, escolhe-se ainda alguma medida de desempenho, como a taxa de erro da rede durante a fase de validação – onde um conjunto de validação, nunca apresentado a rede, é utilizado para avaliar seu desempenho – considerando um limiar que será validado por um número máximo de repetições. Se o limiar não for atingido, o treinamento cessa[6].

2.3.4 OTIMIZAÇÃO BASEADA EM GRADIENTE

Várias tarefas em aprendizado de máquina podem ser definidas como um problema de otimização. Comumente, o objetivo é a minimização de uma função de custo ou erro: realizando o cálculo do gradiente da função revela em qual direção a função decresce mais rapidamente, sendo então possível dar passos em direção a um mínimo local. Os parâmetros desta função de custo então são atualizados na direção oposta a do gradiente calculado para a função em relação aos seus parâmetros. No **Gradiente Descendente** tradicional, o tamanho deste passo é chamado de taxa de aprendizado, parâmetro este que deve ser configurado manualmente. A convergência do gradiente descendente é garantida em algumas circunstâncias, porém, o método depende da escolha de uma taxa de aprendizado apropriada bem como a quantidade certa de iterações no conjunto de dados de treinamento. Se a taxa de aprendizado configurada for pequena demais, exigirá muitas épocas de iteração no conjunto de dados de treinamento, se for grande demais, pode causar oscilações na rede neural, dificultando a convergência (RUDER, 2016; ANDRYCHOWICZ et al., 2016; MURPHY, 2012).

[6]em inglês este procedimento é conhecido como *Early stop* ou *Early Stopping*

A versão estocástica desta técnica – **Gradiente Descendente Estocástico**, SGD – é uma simplificação da técnica anterior, sendo menos custosa. Neste método, ao invés de calcularmos o gradiente para todas as amostras do conjunto de dados de treinamento, o calculamos apenas para uma amostra do conjunto, escolhida randomicamente. Isto significa dizer que a atualização dos parâmetros ocorre a cada amostra observada. Portanto, uma época no SGD equivale a uma iteração por todo conjunto de dados. Uma outra variante do SGD utiliza-se de subconjuntos de amostras também selecionadas randomicamente do conjunto de treinamento. A este subconjunto dá-se o nome de *mini-batch*. Cada atualização utilizando a taxa de aprendizado vai ocorrer ao fim do processamento deste subconjunto. Desta maneira, reduz-se a variância na atualização dos parâmetros o que pode levar a uma convergência mais estável (RUDER, 2016; MURPHY, 2012).

Existem outros otimizadores que tentam solucionar os problemas enfrentados pelo SGD, como por exemplo, a escolha da taxa de aprendizado. Para uma lista concisa desses métodos, consulte (RUDER, 2016).

2.3.5 FUNÇÕES DE ATIVAÇÃO

Para que seja calculado o gradiente para cada neurônio de uma rede neural necessita-se calcular a derivada da função de ativação associada a cada neurônio. Para que o cálculo seja possível, a função de ativação deve ser contínua e diferenciável. Analisaremos nesta seção, três das funções mais utilizadas na construção de redes neurais: A **sigmoid**, a **tangente hiperbólica** e a **unidade retificada linear**.

A função **Logística** ou **Sigmoid** é uma função não-linear em formato de "s" (Figura 2.6). Pode ser definida como uma função estritamente crescente – crescendo monotonicamente com sua entrada – que exibe um equilíbrio no comportamento linear e não linear (MITCHELL, 1997; HAYKIN et al., 2009). Como esta função mapeia um grande domínio de entrada para um pequeno intervalo de saídas, é corriqueiramente chamada de função de achatamento[7]. A função logística é definida como

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.8}$$

podendo assumir um intervalo de valores entre 0 e +1. Sua derivada pode ser definida em termos de sua própria saída: $f'(x) = f(x)(1 - f(x))$.

Outra função comumente categorizada como parte da "família sigmoid" é a função **Tangente Hiperbólica** (Figura 2.7). Essa função compartilha características semelhan-

[7]em inglês, *squashing functions*

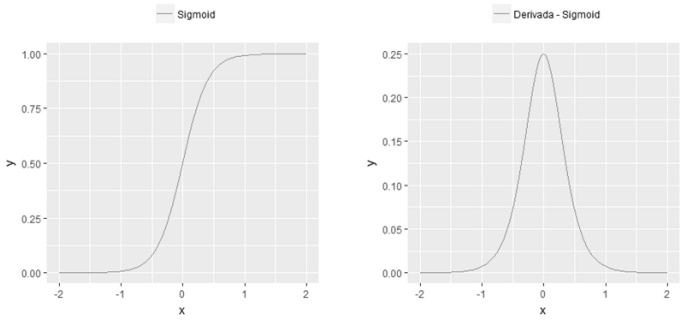

FIG. 2.6: Função sigmoid e sua derivada

tes as da Sigmoid Logística, sendo distinguida apenas por seu intervalo de saída, que varia de -1 até +1 – enquanto a logística varia de 0 a 1. É uma função não linear, contínua e diferenciável em todos os pontos e pode ser definida pela equação

$$f(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{2.9}$$

e sua derivada como $f'(x) = 1 - f(x)^2$. Por também reduzir um domínio de entrada infinito a um intervalo de saída finito também é conhecida como uma função de achatamento (GRAVES, 2012).

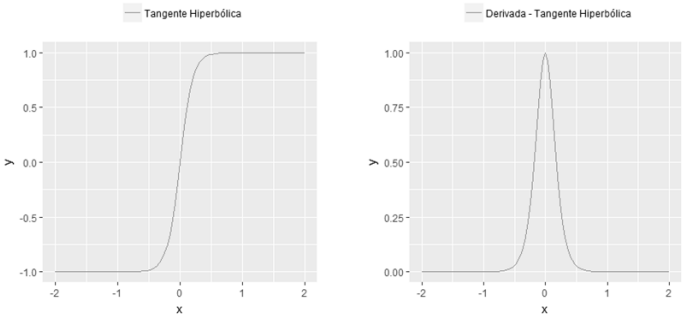

FIG. 2.7: Função Tangente Hiperbólica e sua derivada

De acordo com Goodfellow et al. (2016), a a função **Unidade Linear Retificada** – ReLU – é a função de ativação padrão recomendada para o uso na maioria das redes neurais de alimentação direta. Apesar de ser não linear, mantém-se muito próxima do linear e por isso conserva muitas das propriedades que fazem modelos lineares fáceis de otimizar com métodos baseados em gradiente. Sua equação é definida como

$$f(x) = \begin{cases} 0 \ se \ x < 0 \\ x \ se \ x \geq 0 \end{cases} \tag{2.10}$$

ou simplesmente $f(x) = max(0, x)$, e sua derivada $f'(x) = \begin{cases} 0 \ se \ x \leq 0 \\ 1 \ se \ x > 0 \end{cases}$

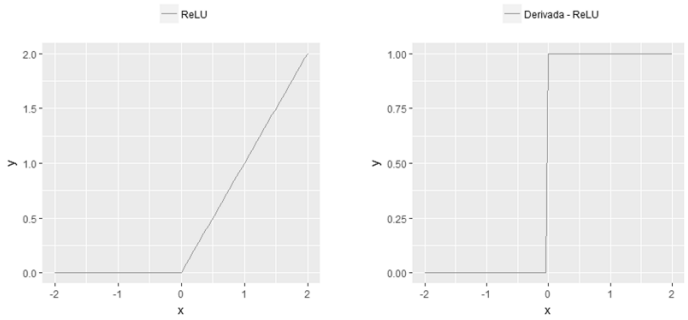

FIG. 2.8: Função ReLU e sua derivada

2.4 APRENDIZADO PROFUNDO

O Aprendizado de Máquina está presente em diversas aplicações que utilizamos em nosso cotidiano, como buscadores na internet, filtros de spam em e-mails, redes sociais, sistemas de segurança, como anti-virus, sistemas de recomendação de produtos em sites de comércio eletrônico, em sistemas de tradução automática, reconhecimento de faces, câmeras e *smartphones*, dentre outros. Algoritmos de aprendizado de máquina tradicionais são limitados em sua capacidade de processar dados naturais em sua forma mais bruta. Para isso, comumente necessitam de criteriosa extração, seleção e transformação dos dados, além de um razoável conhecimento prévio no domínio do problema para que os dados brutos sejam transformados em representações internas ou em vetores de características de forma a serem utilizados pelo algoritmo de aprendizado para alcançar o sucesso em uma determinada tarefa (LECUN et al., 2015).

A dependência nas representações é um fenômeno que aparece por toda a ciência da computação e até mesmo no nosso dia-a-dia. Para muitas tarefas, fica difícil definir quais características devem ser selecionadas e quais são as mais relevantes para a solução de um problema. Uma das soluções para o problema do Aprendizado de Representações está em utilizar o Aprendizado de Máquina para que se descubra essas representações de maneira automática, isto é, para que não apenas se crie um mapeamento entre a representação e a saída de um modelo, mas que também se descubra a própria representação. Desta maneira, sistemas de inteligência artificial poderiam aprender tarefas com o mínimo de intervenção do ser humano (BENGIO et al., 2013; GOODFELLOW et al., 2016).

Métodos de Aprendizado Profundo são métodos de aprendizado de representações em múltiplos níveis, obtidos através da composição de módulos não lineares que transformam

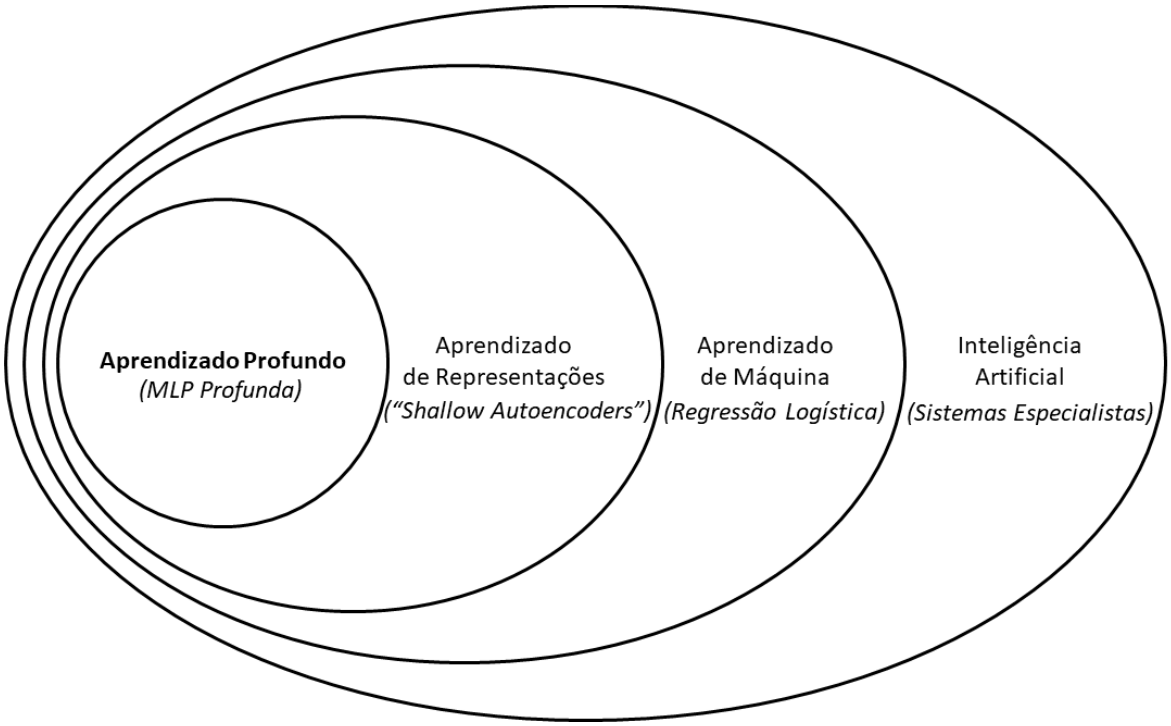

FIG. 2.9: Diagrama de Venn demonstrando onde o Aprendizado Profundo está inserido no contexto da inteligência artificial. Cada seção do diagrama inclui um exemplo de técnica associada. Adaptada de Goodfellow et al. (2016)

representações em um nível mais baixo – começando com a entrada bruta – em uma representação mais abstrata em um nível mais alto. Com a composição de diversas dessas transformações, funções de alta complexidade podem ser aprendidas. E essa é a essência do Aprendizado Profundo: computar representações hierárquicas advindas de dados observacionais, onde características ou fatores de alto nível são definidas com base nos de nível mais baixo (DENG et al., 2014; LECUN et al., 2015).

Um aspecto importante a respeito do aprendizado profundo é o conceito das representações distribuídas. Deng et al. (2014) as descreve como representações internas de dados observados de tal maneira que são modelados como sendo explicados pelas interações de muitos fatores ocultos. Um fator em particular aprendido de configurações de outros fatores geralmente consegue generalizar bem em novas configurações. Este tipo de representação ocorre naturalmente em redes neurais "conexionistas", onde um conceito é representado por um padrão de atividade através de um número de neurônios ou unidades, e onde, ao mesmo tempo, uma unidade tipicamente contribui com muitos conceitos. Uma vantagem deste tipo de representação é que essa correspondência de muitos-para-muitos é o que provê a robustez em representar a estrutura interna dos dados em termos de degradação e a resistência a danos. Outra vantagem é que elas facilitam generalizações

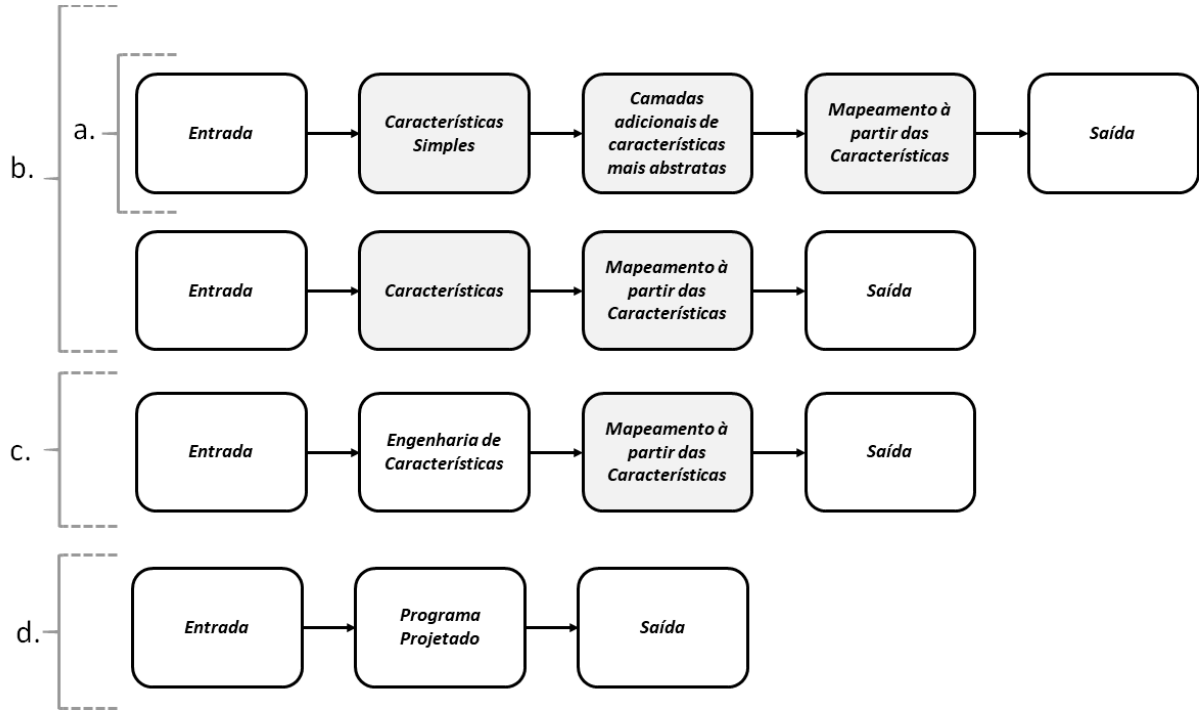

FIG. 2.10: Gráfico mostrando como partes de um sistema de inteligência artificial se relacionam em diferentes disciplinas de inteligência artificial. Em **a.** Aprendizado Profundo, **b.** Aprendizado de Representação, **c.** Aprendizado de Máquina Tradicional e **d.** Sistemas Especialistas. As caixas sombreadas representam componentes que são capazes de aprenderem à partir dos dados. Adaptada de Goodfellow et al. (2016)

de conceitos e relações, possibilitando a capacidade de raciocínio.

Essa série de conceitos teóricos justificam porque, nos últimos anos, técnicas de aprendizado profundo obtiveram destaque na resolução de diversos problemas de computação considerados complexos, superando em muitos casos o estado da arte até então estabelecido e em outros superando até a acurácia humana. Outros fatores também foram determinantes para que só recentemente o Aprendizado Profundo se tornasse viável e tenha obtido tantos resultados favoráveis, como por exemplo: O ***aumento no poder computacional***, como a melhoria da capacidade de processamento das CPUs e o desenvolvimento e amplo uso de GPUs no treinamento das redes neurais, o aumento considerável na disponibilidade e no tamanho de ***conjuntos de dados para treinamento***, e o ***investimento em pesquisa***, com o desenvolvimento de novas técnicas e arquiteturas (DENG et al., 2014; GOODFELLOW et al., 2016).

2.5 ARQUITETURAS DE APRENDIZADO PROFUNDO

Deng et al. (2014) classifica as arquiteturas de aprendizado profundo em três categorias:

Redes neurais profundas **geradoras ou para aprendizado não-supervisionado**, Redes neurais profundas **para aprendizado supervisionado ou discriminativo** e as **redes híbridas**.

As redes neurais profundas **geradoras ou para aprendizado não supervisionado** têm como propósito a captura de correlações de alta ordem à partir de dados observados com o objetivo de realizar uma análise do padrão ou proposta de síntese, quando nenhuma informação sobre os rótulos da classe alvo estão disponíveis.

As redes de aprendizado profundo para o **aprendizado supervisionado**, possuem como objetivo o provimento de poder discriminativo com o propósito de classificação de padrões. Os rótulos dos dados são sempre disponíveis de forma direta ou indireta. Também são chamadas de redes discriminativas profundas.

Já as redes *híbridas* também tem como objetivo o provimento de poder discriminativo, porém, que seja assistida por redes geradoras ou não supervisionadas. Isso pode ser alcançado por melhor otimização ou regularização da rede profunda. O objetivo também pode ser alcançado quando os critérios discriminativos para o aprendizado supervisionado são usados para estimar parâmetros de qualquer rede profunda geradora ou não-supervisionada.

No contexto desta pesquisa, serão discutidas algumas arquiteturas de redes neurais profundas, porém, o foco desta sessão será nas redes Autocodificadoras, por terem conexão direta com este trabalho.

2.5.1 REDES NEURAIS CONVOLUCIONAIS

Redes Neurais Convolucionais são redes inspiradas na estrutura do modelo do sistema visual proposto por Hubel e Wiesel (1962), que descobriram a organização hierárquica do córtex visual de mamíferos, onde grupos de neurônios ligados diretamente às retinas são responsáveis por detectarem padrões simples – como arestas e bordas – e outros grupos hierarquicamente superiores detectam padrões mais complexos baseados em padrões previamente detectados. As primeiras redes convolucionais[8], que foram desenvolvidas por LeCun (LECUN et al., 1989, 1998), já eram treináveis pelo gradiente do erro e alcançam o estado da arte em problemas de visão computacional. Redes convolucionais são redes neurais especializadas no processamento de dados no formato de vetores multidimensionais e são organizadas tradicionalmente em sequências de três tipos de camadas: Convoluci-

[8]Outra arquitetura de rede foi desenvolvida baseada nos trabalhos de Hubel e Wiesel (1962) foi a Neocognitron (FUKUSHIMA; MIYAKE, 1982). Os filtros ou campos receptivos da Neocognitron não eram treináveis e sim previamente programados.

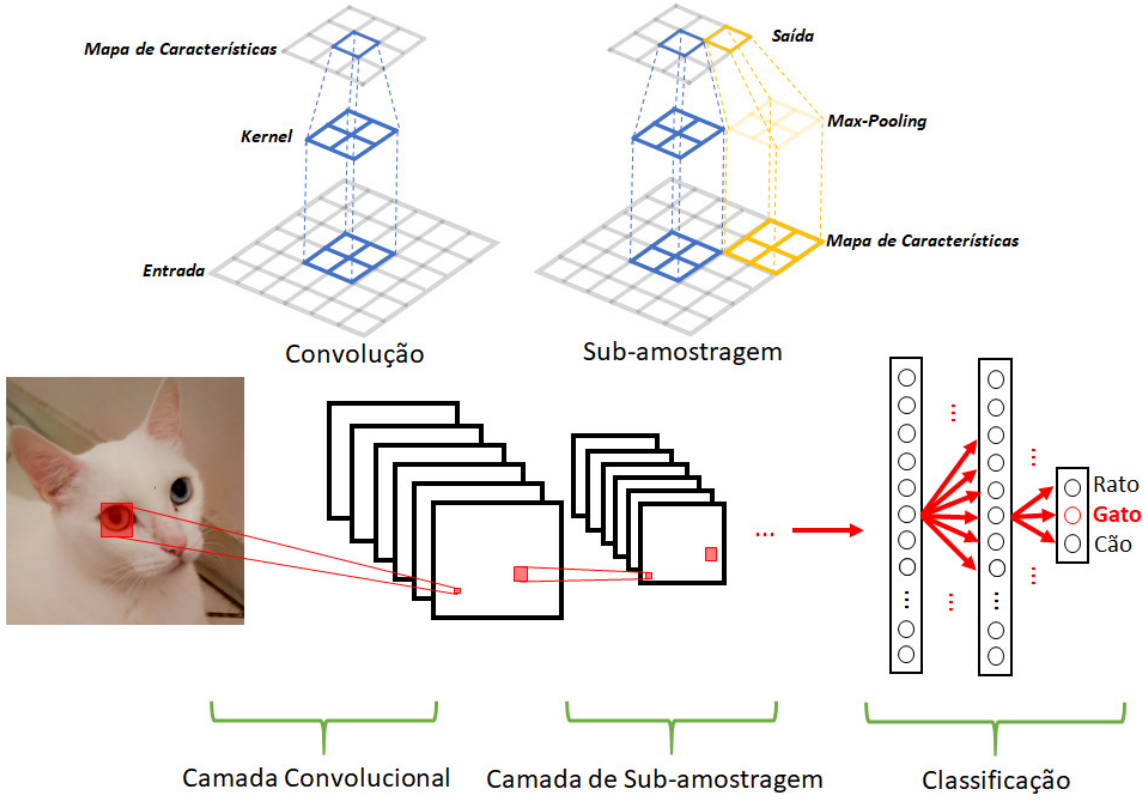

FIG. 2.11: Uma Arquitetura de uma Rede Neural Convolucional comum. Geralmente, essas redes são formadas por uma camada convolucional, seguida de uma camada de sub-amostragem. Essas duas camadas se repetem até que camadas com neurônios completamente conectados são adicionados à arquitetura da rede com o objetivo final de realizar uma classificação do objeto de entrada. Adaptada de Guo et al. (2016)

.

onais, Subamostragem e Camada de neurônios completamente conectados (Figura 2.11) (BENGIO et al., 2009; GOODFELLOW et al., 2016).

Na camada convolucional, um **filtro** ou **kernel** desloca-se por toda a região dos dados de entrada. Na sobreposição entre o filtro e uma região dos dados de entrada é então calculado um produto escalar, cujo resultado é a extração de uma característica para esta sub-região. Ao fim deste processo, teremos então um mapa de características extraídas a partir de um determinado filtro (figura 2.12). A dimensionalidade do filtro e o valor do deslocamento influenciam diretamente no tamanho resultante do mapa de características. A quantidade de filtros determina a quantidade de mapas de características gerados pela camada convolucional (BENGIO et al., 2009; GOODFELLOW et al., 2016).

Outras características motivam o uso de convoluções, já que contribuem para a cons-

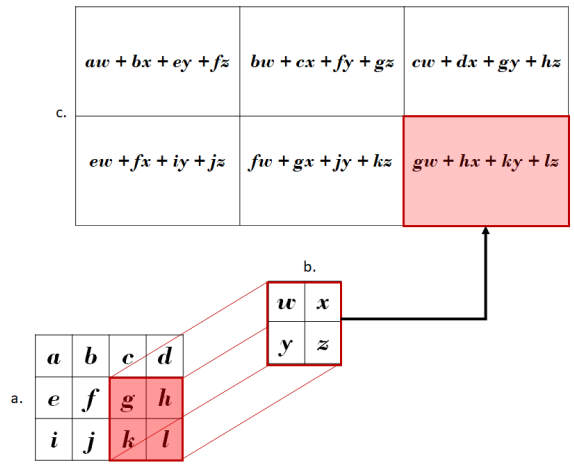

FIG. 2.12: Convolução 2D do tipo "válida", do inglês *"valid"*, onde o filtro ou kernel obedece a delimitação da dimensão dos dados de entrada). Em **a.** podemos observar a matriz de entrada, cuja sub-região em destaque está sofrendo uma convolução com o filtro ou kernel; **b.** é o filtro ou kernel utilizado na operação e **c.** o mapa de características gerado na saída da camada de convolução. Podemos observar qual seria o calculo da convolução para cada unidade do mapa de características. Adaptada de Goodfellow et al. (2016)

trução de bons sistemas de aprendizado de máquina. Convoluções utilizam **conectividade local**[9] (figura 2.13), isso faz com que a rede seja capaz de detectar padrões ou características em espaços menores, com menos parâmetros para serem treinados e consequentemente reduzindo o consumo de memória e o custo computacional do modelo[10]. Outra característica importante é a utilização de pesos compartilhados[11] o que significa dizer que um mesmo parâmetro para mais de uma função em um modelo. Em uma rede neural tradicional cada elemento da matriz de pesos é usada exatamente uma vez quando calculamos a saída de uma camada. Em uma rede convolucional, cada membro de um kernel é usado em cada posição dos dados de entrada. Isso significa dizer que em uma convolução, ao invés de se aprender um conjunto de parâmetros para cada localidade dos dados de entrada, o modelo aprende apenas um conjunto de parâmetros que será compartilhado. A outra característica diz respeito a capacidade de reconhecer padrões aprendidos em uma sub-região dos dados de entrada em outras regiões, ou seja, o que um kernel ou filtro aprendeu em uma região pode ser reconhecido em qualquer outra região, o que torna a representação dos dados **equivariante a translações**.

[9] Interações Esparsas, Conectividade Esparsa, Pesos Esparsos também são sinônimos de conectividade local

[10] Ao contrário de um modelo de rede onde os neurônios são completamente conectados, ou seja, um neurônio de uma camada $x+1$ está conectado a todos os neurônios da camada anterior, x, por exemplo.

[11] também conhecido como pesos amarrados, do inglês *tied weights*.

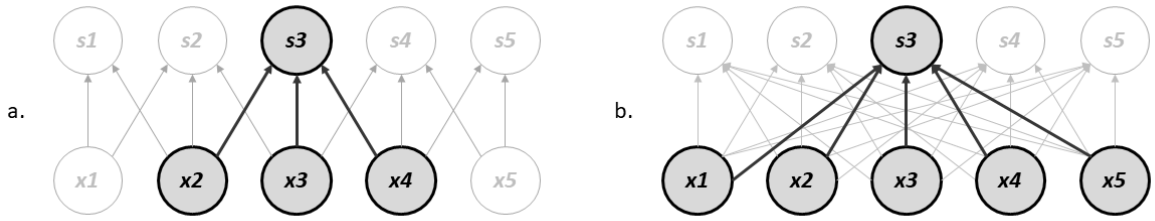

FIG. 2.13: Em **a.** Conectividade local. Podemos observar duas camadas, x e s, sendo s a camada de saída. Em destaque uma unidade $s3$ e as unidades $x2$, $x3$ e $x4$ representando seu campo receptivo. Neste caso, formado por uma convolução de um kernel com tamanho 3, apenas as três unidades em destaque na camada x afetam a unidade $s3$. Em **b.** vemos como seria a interconexão dos neurônios da camada x com o mesmo neurônio $s3$ em um esquema de conectividade de redes neurais tradicionais: toda a camada x afeta no resultado de $s3$. Adaptada de Goodfellow et al. (2016)

.

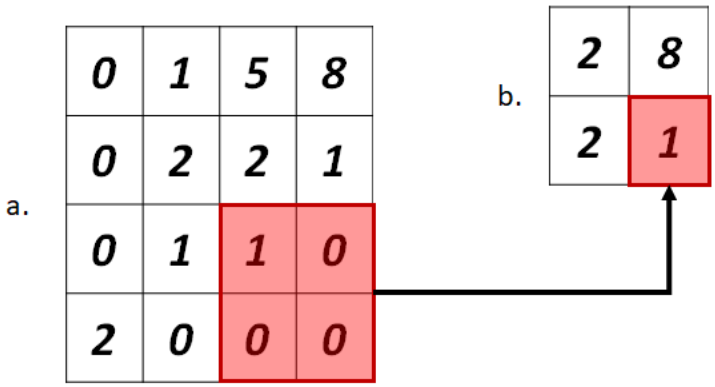

FIG. 2.14: Exemplo da aplicação da subamostragem por *Max-pooling* em um mapa de características. Somente as maiores ativações de cada sub-região são extraídas. Adaptada de (KARPATHY, 2018)

Após a camada convolucional geralmente inclui-se uma camada de subamostragem. A camada subamostragem tem como objetivo principal realizar a redução do número de parâmetros da rede, o que ajuda a evitar o superajustamento do modelo. Uma "janela" desloca-se por toda a região de um mapa de características já ativado, aplicando o cálculo configurado onde houver sobreposição. Como mecanismo de subamostragem, comumente utiliza-se o *Max-pooling*[12], que extrai o valor máximo das ativações em uma sub-região do mapa de características (figura 2.14). Uma propriedade interessante, que pode ser introduzida pelo *Max-pooling*, é que ele possibilita que a rede aprenda representações robustas a pequenas transformações nos dados, como distorções ou translações (invariância de representação)(GOODFELLOW et al., 2016; SCHMIDHUBER, 2015).

Ao fim da rede convolucional, com o objetivo de realizar a tarefa de classificação, inclui-se uma ou mais camadas com neurônios completamente conectados, finalizando assim o que viria a ser uma arquitetura de rede convolucional tradicional.

2.5.2 REDES NEURAIS RECORRENTES

Podemos definir as Redes Neurais Recorrentes – RNN – como um família de redes especializada no processamento de dados sequenciais (GOODFELLOW et al., 2016). Diferentemente de redes neurais *feedforward*, as RNNs permitem a criação de conexões cíclicas entre neurônios. Essa mudança nas conexões faz com que RNNs consigam em princípio mapear todo o histórico de entradas observadas anteriormente para cada saída específica, funcionando como uma espécie de memória gravada nas camadas ocultas da rede, que acabam influenciando nos valores de saída do modelo (GRAVES, 2012).

As sequências processadas por uma RNN podem ou não possuir o mesmo tamanho. Para que consigam generalizar para sequências de diferentes tamanhos em diferentes posições no tempo, esta rede implementa o conceito de compartilhamento de parâmetros. Cada membro da saída da rede é uma função dos membros anteriores da saída. Cada membro da saída é produzido usando a mesma regra de atualização aplicada a saídas anteriores. Essa formulação recorrente resulta no compartilhamento de parâmetros através de um grafo computacional muito profundo (GOODFELLOW et al., 2016).

Operam em uma sequência que contêm vetores $x^{(t)}$, sendo t o índice de um passo no tempo, variando em uma faixa de 1 até τ – consequentemente os vetores vão de $x^{(1)}, \ldots, x^{(\tau)}$. O estado interno ou oculto da rede no decorrer do tempo pode ser definido

[12] Apesar de existirem outros mecanismos de subamostragem, como o de soma dos valores, *Sum-Pooling*, ou o da média, *Mean-Pooling*.

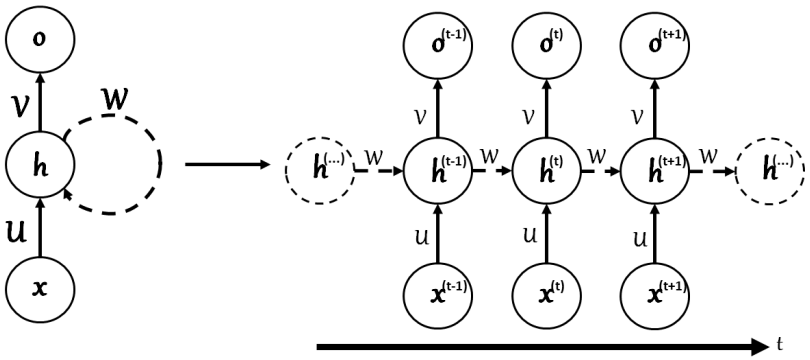

FIG. 2.15: Grafo computacional de uma RNN mapeando a sequência de entrada x para uma sequência de saída o. À esquerda podemos visualizar a rede com suas conexões recorrentes e à direita, a rede desdobrada no tempo, como uma MLP. Podemos observar a conexão entre a entrada e o estado oculto (x-h) parametrizado pela matriz de pesos U, conexões recorrentes de estado oculto para estado oculto (h-h) parametrizados pela matriz de pesos W e conexões de estado oculto para saída (h-o) parametrizados pela matriz de pesos V. Função de custo e a comparação com a classe alvo de saída foram omitidos para manter a simplicidade. Adaptada de (GOODFELLOW et al., 2016).

como

$$h^{(t)} = f(h^{(t-1)}, x^{(t)}; \theta) \tag{2.11}$$

onde $h^{(t)}$ é o vetor que representa o estado oculto da rede no tempo t, $x^{(t)}$ é o vetor de entrada no tempo t e Θ são os parâmetros de f, que permanece constante a passagem de t, isto é, aplica a mesma função com os mesmos parâmetros em cada interação (GOODFELLOW et al., 2016).

Dependendo do arranjo das conexões configuradas em uma RNN podemos ter modelos completamente diferentes, com processamento de entradas e saídas igualmente diversos. Um arranjo mais clássico seria o de uma RNN que produz uma saída a cada passo no tempo e que possuem conexões entre seus estados ocultos. Outra configuração seria a que produz a cada passo no tempo uma saída e possui uma conexão de retorno da camada de saída para a camada oculta da rede. Uma outra configuração possível seria uma rede recorrente que possui capacidade para ler uma sequência de entrada completa e só então ao final de todo processo produzir apenas uma saída.

Apesar de ser uma arquitetura flexível, este tipo de rede enfrenta alguns problemas durante seu treinamento. Um dos grandes desafios ao treinar uma RNN é que elas podem possuir dependências de longo prazo, ou seja, em determinadas circunstâncias, para produzir uma saída no tempo t a rede dependerá de dados processados em seu estado interno muitos passos no passado. O problema dessas dependências de longo prazo é que o cálculo dos gradientes propagado por tantos passos tende a desaparecer ou explodir.

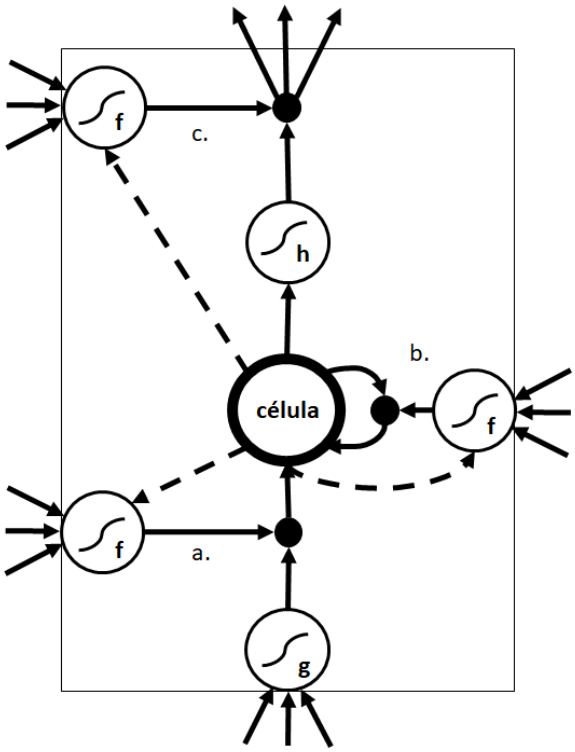

FIG. 2.16: Bloco LTSM com uma célula. As três portas (entrada, saída e esquecimento) são unidades de soma que coletam ativações de dentro e de fora do bloco, controlando a ativação da célula de memória via multiplicações, representadas na imagem por círculos pretos. As portas de entrada (**a.**) e saída (**b.**) multiplicam a entrada e saída da célula de memória, enquanto a porta de esquecimento (**c.**) multiplica com o estado anterior da célula. Nenhuma função de ativação é aplicada dentro da célula. A função f, que são ativações das portas, resultam entre 0 (porta fechada) e 1 (porta aberta). A função de ativação geralmente utilizada como ativação da camada de entrada (g) e saída (h) são a tangente hiperbólica ou a sigmoide logística, contudo, em algumas arquiteturas utiliza-se a função identidade ($f(x) = x$). As linhas tracejadas simbolizam as únicas conexões ponderadas dentro do bloco LSTM. Adaptada de Graves (2012).

A essas duas situações, dá-se o nome de Desaparecimento do Gradiente[13] e Explosão do Gradiente[14]. Uma das soluções para o problema de dependências de longo prazo seria a de criar caminhos no tempo que tenham derivadas que não levem ao desaparecimento nem a explosão dos gradientes. A intuição aqui é que, após uma informação tenha sido utilizada, pode ser útil para a rede neural o **esquecimento de um estado anterior**. Um dos métodos que implementam este conceito são as unidades ou células **Long Short-term Memory** – LSTM (Figura 2.16) (GOODFELLOW et al., 2016). Ao introduzir unidades de controle internas análogas a operações de leitura, escrita e *reset*, permite-se que essas células de memória armazenem e recuperem informações através de longos períodos de tempo, mitigando o problema do desaparecimento do gradiente (GRAVES, 2012).

2.5.3 REDES AUTOCODIFICADORAS

Uma rede Autocodificadora ou Autoassociativa é uma rede de alimentação direta e acíclica, que é treinada de maneira não-supervisionada, possuindo como objetivo a aproximação de uma função identidade – $f(x) = x$, através do mapeamento da dimensão de entrada para uma nova representação. Em outras palavras, a rede procura aprender como copiar os valores dos neurônios de entrada para seus neurônios de saída com o mínimo de perda (GOODFELLOW et al., 2016).

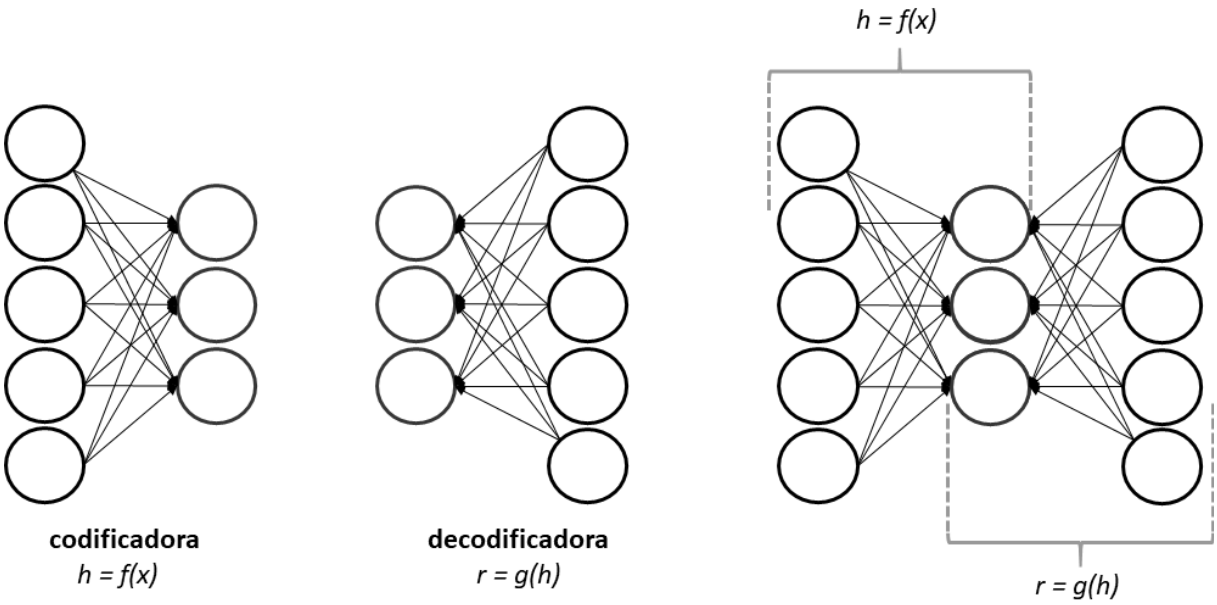

FIG. 2.17: Uma Rede Autocodificadora decomposta nas funções codificadora e decodificadora. Equações de Goodfellow et al. (2016).

[13]em inglês, *Vanishing Gradient Problem*
[14]em inglês, *Exploding Gradient Problem*

A respeito de sua constituição (Figura 2.17), pode ser visualizada como uma junção de dois blocos de redes, duas funções. A primeira, conhecida como ***codificadora***, que pode ser definida como $h = f(x)$, é responsável por codificar os valores dos dados de entrada x para uma nova representação h. A finalidade da ***decodificadora*** é, através da minimização de uma função de custo, realizar uma reconstrução dos dados de entrada x utilizando como entrada a representação contida na camada oculta, isto é, $r = g(h)$. Quanto a sua aplicação, tradicionalmente são empregadas em tarefas de **redução de dimensionalidade** ou no **aprendizado de características** (GOODFELLOW et al., 2016).

Podem também ser classificadas de acordo com o número de neurônios em sua camada oculta, portanto, ***subcompletas*** e ***sobrecompletas***. ***Autocodificadoras subcompletas*** (Figura 2.18) são aquelas cuja quantidade de neurônios de sua camada oculta é menor que a quantidade de neurônios de suas camadas de entrada e saída, formando uma espécie de "gargalo" na rede. Esta configuração de rede é utilizada classicamente para tarefas de redução de dimensionalidade, também conhecida como compressão de dados, onde apenas as características mais importantes serão aprendidas na camada oculta. Já a variante ***sobrecompleta*** (Figura 2.19) ocorre quando a quantidade de neurônios da camada oculta é maior que a quantidade de neurônios da camada de entrada, produzindo, neste caso, uma representação da entrada da rede em uma dimensão superior. Para redes *sobrecompletas*, é habitual o uso de mecanismos de regularização para que se impeça a rede de copiar fielmente os dados apresentados em sua camada de entrada devido ao maior número de neurônios na camada oculta.

No que concerne o ***processo de aprendizado*** de redes autocodificadoras tradicionais, podemos especificá-lo como a minimização de uma função de custo

$$L(x, g(f(x))) \tag{2.12}$$

cujo propósito é o de minimizar o erro de reconstrução, ou seja, a discrepância entre os dados originais da entrada e os dados que foram reconstruídos pela rede. L poderia ser qualquer função de custo, como por exemplo a função de ***MSE – Erro Quadrático Médio*** . A função de custo atua como um elemento penalizador em $g(f(x))$ caso seu resultado seja discrepante de x. Quando uma autocodificadora subcompleta utiliza a função de custo MSE em sua arquitetura, seu decodificador utiliza uma função linear, a rede aprende como efeito colateral o subespaço principal dos dados de treinamento. Isso é equivalente a afirmar que a rede se comporta como um PCA. Contudo, quando esta

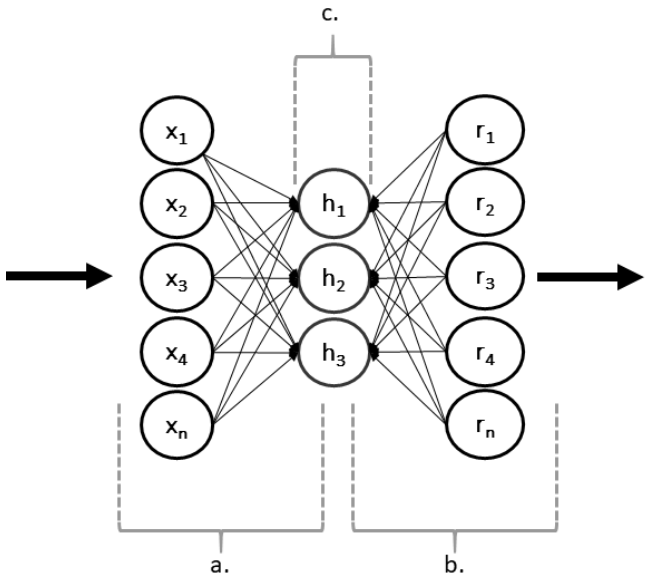

FIG. 2.18: Exemplo de uma **Rede Autocodificadora subcompleta**. É possível visualizar o "gargalo" criado pela redução da dimensionalidade na camada oculta da rede. Podemos igualmente observar: **a.** a função codificadora; **b.** a função decodificadora; e **c.** a camada oculta, a nova representação ou representação latente. As setas indicam a direção do fluxo de informações na rede, da entrada para a saída.

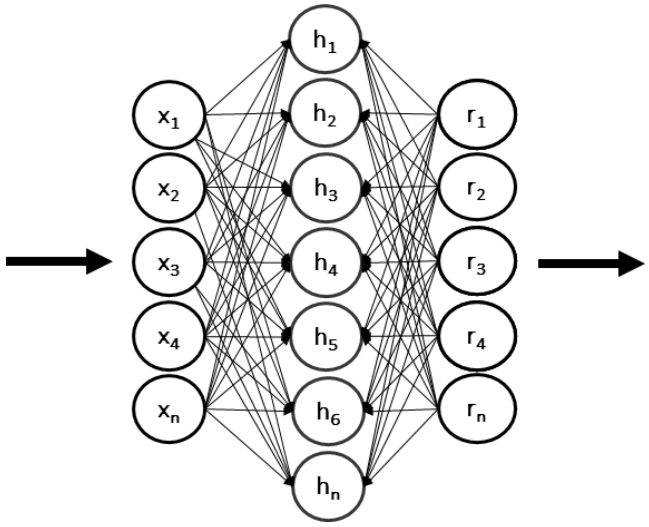

FIG. 2.19: Exemplo de uma **Rede Autocodificadora sobrecompleta**. É possível visualizar a camada oculta com um número de neurônios maior do que as camadas de entrada ou saída, o que significa dizer que esta rede faz um mapeamento de características para uma dimensão superior. As setas indicam a direção do fluxo de informações na rede, da entrada para a saída.

rede é construída utilizando-se de funções não lineares tanto para sua função codificadora quanto para a decodificadora, diz-se então que a rede aprendeu uma generalização mais poderosa do PCA (GOODFELLOW et al., 2016).

2.5.3.1 AUTOCODIFICADORAS REGULARIZADAS

Alguns problemas podem surgir ao se treinar redes autocodificadoras no que diz respeito a sua capacidade. Sabe-se que essas redes podem falhar em aprender sua tarefa caso seu codificador e decodificador possuírem grande capacidade. O mesmo problema também ocorre quando a camada oculta possui a uma dimensionalidade maior ou igual aos dados de entrada da rede – caso sobrecompleto – nesses casos, a rede pode aprender a copiar os dados de entrada para a saída da rede, sem necessariamente ter aprendido coisa alguma sobre a distribuição dos dados observados. Alguns mecanismos de regularização permitem que as redes autocodificadoras possam ser treinadas sem a restrição da dimensionalidade imposta pelo aprendizado em uma rede subcompleta e até mesmo sem a restrição de profundidade da rede, isto é, sem a restrição de sua capacidade. Consequentemente, ao invés de limitar sua capacidade, autocodificadoras regularizadas utilizam funções de custo que forçam os modelos gerados a perseguirem outras propriedades além da simples cópia de sua entrada para sua saída, como por exemplo, a esparsidade de sua representação ou a robustez a ruídos ou ausência de dados (GOODFELLOW et al., 2016).

A primeira variante que abordaremos são as ***Autocodificadoras Esparsas***[15]. Uma Autocodificadora Esparsa é uma rede com a mesma arquitetura da autocodificadora tradicional, contudo, seu treinamento agora inclui uma penalização a esparsidade – $\Omega(h)$ – na camada de codificação h em adição ao erro de reconstrução já visto anteriormente, portanto:

$$L(x, g(f(x))) + \Omega(h) \tag{2.13}$$

onde $g(h)$ é a saída da função decodificadora e $h = f(x)$ é a saída da função de codificação. Essa variação é tipicamente utilizada para o aprendizado de características com o intuito de serem utilizadas em outras tarefas, como por exemplo, tarefas de Classificação. Ao ter sido regularizado para se tornar esparso, uma autocodificadora deve responder a características estatísticas intrínsecas do conjunto de dados ao qual foi treinada, ou seja, além de ser treinada para realizar a cópia dos dados de entrada para a saída da rede, a penalização de esparsidade pode gerar um modelo que aprendeu características úteis como um subproduto do treinamento com a penalização (GOODFELLOW et al., 2016).

Uma ***Autocodificadora Extratora de Ruídos***[16] – DAE – tem como objetivo produzir representações robustas a ruídos ou mesmo a ausência ou corrupção de dados. Para que isso seja possível, durante seu treinamento, uma cópia do dado de entrada original x

[15]Do Inglês, *Sparse Autoencoder*
[16]do Inglês, *Denoising Autoencoder*

é realizada. Essa cópia passa então por um processo corrupção de dados ou de adição de ruído – $\tilde{x}$. O dado corrompido é utilizado como entrada da rede. A tarefa que deverá ser aprendida pela rede é a de reconstruir o dado original a partir de um dado corrompido, ou seja:

$$L(x, g(f(\tilde{x}))) \tag{2.14}$$

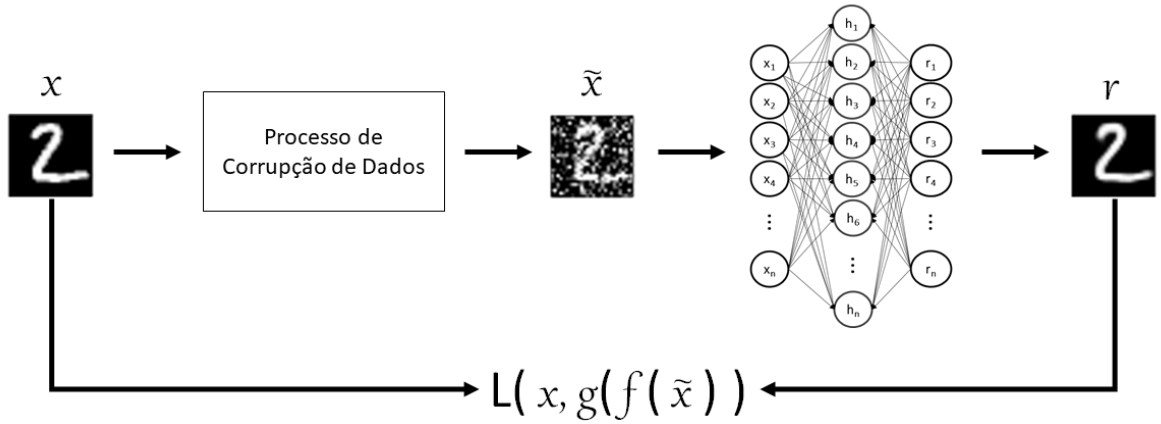

FIG. 2.20: Exemplo hipotético do processo de treinamento de uma ***Autocodificadora Extratora de Ruídos*** com dados do MNIST. Observa-se que o processo de aprendizado começa com o dado original passando por um processo de corrupção dos dados, neste caso foi aplicado o ruído branco gaussiano aditivo (AWGN). O dado corrompido é apresentado a rede e a saída produzida é comparada com o dado original, medindo-se assim o erro de reconstrução. Equação de (GOODFELLOW et al., 2016)

2.5.3.2 AUTOCODIFICADORAS PROFUNDAS

Uma rede autocodificadora é constituída de duas partes, seu codificador e o seu decodificador, que tradicionalmente são construídos com uma única camada oculta. Contudo, existe a possibilidade de se criar uma autocodificadora profunda, isto é, uma rede com um número de camadas ocultas maior do que um (DENG et al., 2014; GOODFELLOW et al., 2016).

Existem algumas vantagens ao utilizar uma arquitetura profunda. O codificador e o decodificador, individualmente, são redes de alimentação direta, o que traz para a Autocodificadora todas as vantagens que redes de alimentação direta possuem em relação a profundidade da rede. Uma das vantagens trazidas pela profundidade em redes de alimentação direta, está relacionada ao Teorema de Aproximação Universal. Este teorema garante que uma rede neural de alimentação direta com ao menos uma camada oculta

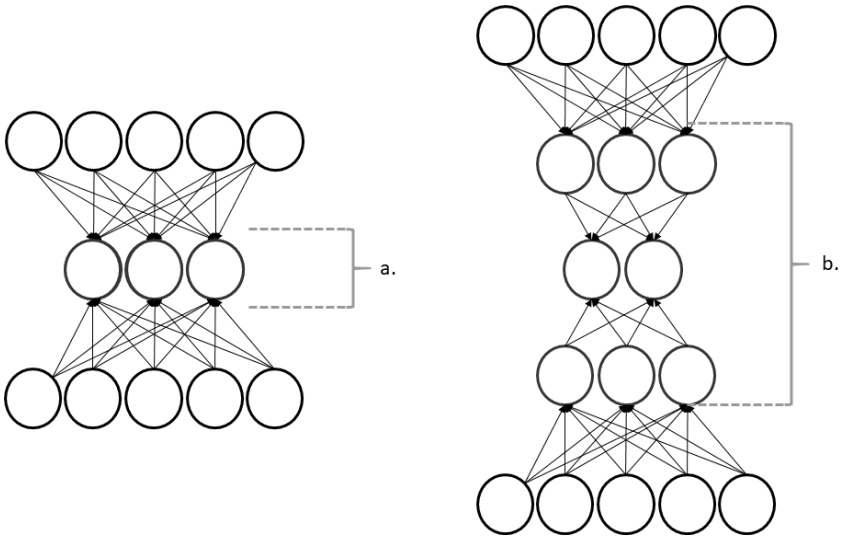

FIG. 2.21: Comparação entre uma Rede Autocodificadora Superficial e uma Rede Autocodificadora Profunda. Podemos observar em **a.** uma arquitetura superficial com apenas uma camada oculta e em **b.**, uma arquitetura profunda com três camadas ocultas.

pode representar uma aproximação de qualquer função – pertencente a uma classe ampla de funções – com um grau de acurácia arbitrário, contanto que existam unidades ou neurônios suficientes na camada oculta. Isso significa dizer que uma rede autocodificadora é capaz de representar a função identidade com apenas uma camada oculta de maneira satisfatória. Contudo, o mapeamento dos dados de entrada utilizando apenas uma camada oculta é superficial. Devido a esta superficialidade, não é possível forçar restrições arbitrárias a rede, como por exemplo, que a representação gerada na camada oculta seja esparsa. A profundidade também pode reduzir exponencialmente o custo computacional de representar determinadas funções, da mesma maneira que pode também reduzir exponencialmente a necessidade da quantidade de dados de treinamento necessários para aprender algumas destas funções (GOODFELLOW et al., 2016).

Quanto ao treinamento de uma rede autocodificadora profunda, podemos destacar que existem duas maneiras de realizá-lo. A primeira, consiste no pré-treinamento não supervisionado em camadas, também conhecido como empilhamento – *stacking* – que consiste no pré-treinamento de redes com apenas uma camada oculta, onde as características aprendidas por uma das redes alimenta a entrada da próxima e assim sucessivamente, para ao final, formarem uma autocodificadora profunda que é novamente treinada via *backpropagation* (HINTON; SALAKHUTDINOV, 2006; ERHAN et al., 2010). O treinamento de redes profundas passou por mudanças a partir do momento em que novas técnicas de regularização foram adotadas, como por exemplo, o uso de ReLU (GLOROT et al., 2011), dentre outras, fazendo com que a técnica do empilhamento fosse deixando de ser

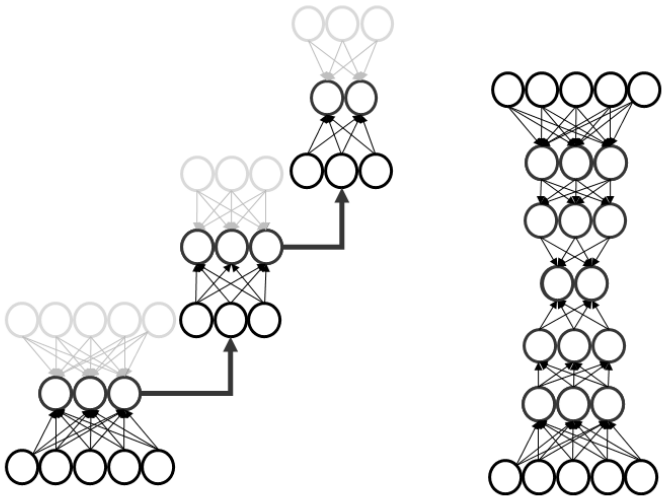

FIG. 2.22: Exemplo hipotético do pré-treinamento não supervisionado ou empilhamento para criação de uma rede autocodificadora profunda. Adaptada de (HINTON; SALAKHUTDINOV, 2006)

obrigatória para o treinamento de modelos profundos.

2.5.3.3 OUTRAS VARIANTES

Podemos ainda citar outras variantes às redes autocodificadoras já descritas nesta seção, como por exemplo, a ***rede autocodificadora contrativa*** (RIFAI et al., 2011) que introduz um novo mecanismo de regularização, possibilitando a rede a gerar representações invariantes a pequenas mudanças nos dados de treinamento. A ***rede autocodificadora extratora de ruído empilhada*** (VINCENT et al., 2010) que utiliza DAEs com a técnica de treinamento empilhado com o objetivo de criar representações de alto nível que melhoraram resultados de alguns classificadores. Uma outra variante, categorizada como uma rede do tipo geradora é a ***rede autocodificadora variacional***, que aprende a distribuição dos dados de treinamento e a partir disso é capaz de gerar novos exemplos de dados. Isso acontece devido a algumas características importantes, dentre elas podemos destacar o uso do espaço latente contínuo o que permite a amostragem randômica e a interpolação de dados (DOERSCH, 2016; REZENDE et al., 2014; KINGMA; WELLING, 2013).

2.5.4 COMPARANDO ARQUITETURAS

As arquiteturas de redes neurais apresentadas em tópicos anteriores são completamente distintas umas das outras. Apesar de ser possível criar uma rede com camadas mistas,

por exemplo, uma Autocodificadora Convolucional ou uma LSTM Convolucional, iremos aqui apenas considerar as arquiteturas tradicionalmente utilizadas, conforme descrito nas seções anteriores.

De uma forma geral, independente da topologia escolhida, o treinamento de redes neurais implica na escolha e ajuste de uma série de parâmetros que influenciam diretamente no sucesso ou fracasso de um modelo. Podemos citar como parâmetros em comum que dessas redes:

a) Otimizador e possíveis parâmetros. Por exemplo, a taxa de aprendizado do Gradiente Descendente Estocástico, ou o termo de *momentum* em uma outra variante;

b) O Número de Épocas ou Iterações de Treinamento;

c) Tamanho do Batch, isto é, a quantidade de amostras que serão propagadas pela rede;

d) Funções de Ativação;

e) Número de camadas ocultas;

f) Número de neurônios por camada;

g) Critério de inicialização dos pesos da rede;

h) Critério de Regularização;

i) Função de Custo/Erro

As Redes Recorrentes são as mais versáteis em relação ao tipo de dado processado. Outras arquiteturas de redes neurais processam vetores de entrada de tamanho fixo e igualmente produzem vetores de saída com tamanho fixo. RNNs são interessantes por não estarem presas a esses limites: possuem a capacidade para o processamento de sequências de vetores de entrada de tamanhos diversos e produzir sequências de vetores de saída de tamanhos distintos. A grande desvantagem está relacionada também com sua grande vantagem, a memória: sequências longas demais podem levar ao problema do Desaparecimento do Gradiente, que pode ser diminuído com o uso de mecanismos de regularização ou a utilização de células de memória especializadas, como as LSTMs (GRAVES, 2012; GOODFELLOW et al., 2016).

Apesar dos grandes resultados alcançados pelas Redes Convolucionais, estas ainda possuem muitos outros parâmetros a serem configurados: no que diz respeito ao *kernel*,

devemos definir suas dimensões, deslocamento, o tipo de convolução; se utilizará *zero padding*; qual o critério de subamostragem, as dimensões da janela de subamostragem e o seu deslocamento. Além da grande quantidade de parâmetros a serem definidas e ajustadas, as CNNs possuem outras vantagens além das já mencionadas em seções anteriores: Os dados utilizados pela rede podem ser de diversas dimensionalidades e utilizarem vários canais, onde cada canal é uma observação de uma quantidade em algum ponto no espaço ou tempo. Por exemplo, para uma rede, pode ser que seja necessário processar dados em 2-D, como uma imagem. Ao operar em um único canal, teríamos o equivalente a uma imagem em escala de cinza. Já operando em multicanais, teríamos um canal para cada cor, ou seja, a mesma imagem representada em 3 matrizes, uma para os *pixels* vermelhos, uma para os verdes e outra para os azuis. Uma desvantagem desta rede é que, uma vez que a dimensão dos dados de entrada e saída sejam definidas, não se pode mais alterá-las (GOODFELLOW et al., 2016).

Chegamos então nas Redes Autocodificadoras. Essas redes são de simples treinamento, não necessitando de mais ajustes do que já podemos realizar em uma rede *feed-forward*, como uma MLP. Contudo, está limitada em relação aos dados tanto de entrada como de saída: são fixos, uma vez definidos, o modelo não consegue operar com dados de dimensionalidades diferentes (GOODFELLOW et al., 2016).

3 ANÁLISE DE MALWARE

A análise de *malware* é o processo pelo qual um código potencialmente malicioso é detectado e classificado, tendo seu potencial de impacto descoberto. Detectar e classificar código malicioso são tarefas desafiadoras, geralmente morosas e meticulosas, que necessitam de um especialista que empregará diversas técnicas e ferramentas para realizar a análise de um artefato.

Desenvolvedores de *malware* frequentemente aumentam seus códigos com procedimentos de evasão que emulam programas inócuos ou se utilizam de outras técnicas, como ofuscamento de código, código polimórfico, criptografia, empacotamento, dentre outras; criando óbices ao processo investigatório.

Nas seções subsequentes, definiremos o que é um *malware*, destacando algumas de suas famílias e suas características, potenciais de dano e métodos de evasão comumente empregados, bem como as técnicas utilizadas para sua análise.

3.1 MALWARE: TERMINOLOGIA E TAXONOMIA

Desde os primeiros trabalhos teóricos, que procuraram formalizar e experimentar o que viria a ser conhecido como um vírus de computador (COHEN, 1987) e uma classe mais ampla de programas infecciosos (ADLEMAN, 1990), descrevendo seu comportamento, potencial para danos, sua capacidade de infecção, modelos de prevenção e mecanismos de proteção e sua complexidade de detecção, que essas ameaças evoluíram, ganhando novos objetivos e novas ferramentas e explorando vulnerabilidades em um sempre crescente número de vetores de contágio, tornando-se a ferramenta predominante na prática de crimes cibernéticos.

Um *MALWARE*, *"**MAL**icious Soft**WARE**"* ou Código Malicioso, pode ser definido como qualquer código cujo objetivo seja o de causar dano ou de subverter alguma função do sistema ao qual ele esteja inserido, frequentemente sem a autorização do proprietário (MCGRAW; MORRISETT, 2000). Propagam-se beneficiando-se de vulnerabilidades em ***hardware*** – *hardware trojans*, clones ilegais, ataques de canais laterais; **redes** – ataques baseados em protocolos, monitoramento e *sniffing* de redes; em **aplicações e sistemas operacionais** – *bugs* de gerenciamento de memória, *buffer overflow*, validação de dados de entrada de usuários, privilégios de acesso de usuários, condições de corrida, dentre

outros; e até mesmo utilizando-se de **características peculiares de novas tecnologias**, como redes sociais (DAMSHENAS et al., 2013; JANG-JACCARD; NEPAL, 2014; TEHRANIPOOR et al., 2017).

São difíceis de ser detectados e sua classificação é complexa, visto que a fronteira entre as famílias de *malware* torna-se cada vez mais obscura, dado que as versões mais atuais desses programas aparentam ser uma forma híbrida dentre tais famílias (MCGRAW; MORRISETT, 2000; ENISA, 2018). Apesar das dificuldades inerentes a essa taxonomia, pode-se utilizar diversas abordagens para organizar o código malicioso, como por exemplo, do ponto de vista do *ataque*, sendo *segmentado*, para um alvo específico, ou *em massa*; *comportamental*, onde o comportamento exibido pelo programa é o considerado para a classificação; por famílias, focando na autoria e linhagem desses programas, e até mesmo por suas *funcionalidades*, agrupando-os de acordo com as características que esses programas oferecem ao atacante, como por exemplo (ENISA, 2018; PLATFORM, 2018; SIKORSKI; HONIG, 2012):

Um ***vírus*** pode ser definido de maneira simples como um código que possui a capacidade de replicar-se e infectar computadores e arquivos. O contágio de uma máquina alvo depende da execução de um arquivo infectado com o vírus.

Uma ***worm*** é um programa com a capacidade de se auto-replicar e se espalhar por uma rede, infectando assim um número maior de computadores sem qualquer necessidade de interação com um usuário.

Um ***launcher*** possui como objetivo executar outros códigos maliciosos de maneira não trivial, com o objetivo de mante-lo oculto ou de obter acesso privilegiado ao sistema operacional.

Um ***Backdoor*** é um programa que se instala em um computador com o objetivo de possibilitar ao atacante o acesso e execução de comandos remotos de maneira facilitada – com pouca ou sem qualquer autenticação – no sistema infectado.

Um ***Trojan Horse, Trojan*** **ou Cavalo de Tróia** é um termo designado para classificar programas que se utilizam de artifícios para esconder sua verdadeira identidade, como por exemplo, um *malware* embarcado em um pacote que imita um *driver* de rede que foi baixado da *Internet* ou algum software qualquer que execute código de um *backdoor* quando executado pela vítima.

Botnet Drone ou Bot transforma o computador que foi infectado em um participante de uma *Botnet*. O atacante, por meio de um único servidor de comando e controle, consegue enviar ordens para cada participante da rede – normalmente são utilizadas em

ataques em massa, como um DDOS[17].

Um ***Downloader*** é um código cuja existência se resume a realizar *downloads* e instalação de códigos maliciosos em um sistema comprometido. Geralmente são instalados por atacantes assim que conseguem acesso a um sistema.

Já **Malware de roubo de informações** são malware que coletam informações sobre a vítima e comumente as envia para o atacante. Tipicamente é utilizado para obter acesso a contas online de e-mail ou de *internet banking*. Alguns exemplos desta categoria são *sniffers*, *password hash grabbers* e *keyloggers*.

Scareware são projetados para fraudar usuários. Através de mensagens de erro de sistema ou alerta de vírus falsos o usuário é levado a acreditar que comprando e/ou instalando um determinado software irá resolver o problema. Também é utilizado como via de entrada de outros *malware* e no roubo de dados de usuário.

Ransomware são *softwares* maliciosos utilizados no sequestro de dados de um computador alvo; o programa ameaça a destruição ou publicação dos dados com o objetivo de obter lucro através de um resgate.

Rootkits são *malware* que alteram as funcionalidades padrão de um sistema operacional para que possam executar ações maliciosas de maneira oculta, impedindo sua detecção por ferramentas de análise mais simples.

3.2 EVADINDO A DETECÇÃO

Um dos elementos chave para o sucesso de um ataque via *malware* está em sua habilidade de manter-se similar a um programa benigno, ou seja, de manter-se indetectável. Programadores de *malware* utilizam-se de toda uma sorte de técnicas para dificultar o trabalho de um analista. Podemos categorizar diversos destes métodos de evasão em dois grandes grupos – **Ofuscação** e **Anti-Engenharia Reversa** (YOU; YIM, 2010; CHRISTODORESCU; JHA, 2003; BRANCO et al., 2012; SIKORSKI; HONIG, 2012).

3.2.1 OFUSCAÇÃO

De uma maneira geral, as técnicas de ofuscação do código malicioso têm como objetivo dificultar a análise do *malware* desmontado. Isso pode ser alcançado de diversas formas, comumente distorcendo a estrutura sintática do código sem que este perca a coerência semântica. Através da ofuscação de código, busca-se evadir de ferramentas de análise baseadas em detecção de assinaturas. Nesta seção, detalharemos algumas dessas técnicas.

[17] *Distributed Denial of Service* ou Ataque de Negação de Serviço Distribuído

A **encriptação** é uma das técnicas popularmente aplicadas para ofuscação do *malware*. Consiste em encriptar o corpo de código do *malware*, deixando apenas uma rotina decriptadora, responsável pela decifragem do código anteriormente a sua execução. Uma evolução desta técnica aparece nos *malware* oligomórficos, polimórficos e metamórficos. Inicialmente, a encriptação da maneira que era aplicada ainda possuía uma falha. As ferramentas de detecção de *malware* baseadas em assinaturas evoluíram para detectar códigos maliciosos baseados em suas rotinas de encriptação e decriptação, fazendo com que o programador do *malware* tivesse que evoluir a técnica. Um ***Malware* Oligomórfico** consegue realizar mutações em sua própria rotina de decifragem a cada cópia ou evolução. Todavia, devido ao número reduzido ou pré-definido de esquemas de encriptação utilizados, seu código ainda pode ser detectado por assinaturas. A evolução natural seria criar esquemas de encriptação dinâmicos. Com isso surgiram os *engines* **polimórficos**, que empregam diversas técnicas de ofuscação em conjunto para assim, dinamicamente, gerar diferentes esquemas de encriptação a cada evolução. Contudo, alguns *malware* polimórficos conseguem ser detectados por seu comportamento durante a execução em um ambiente controlado, como um *sandbox*. A próxima evolução seriam os chamados ***malware* metamórficos**, programas aptos a reescreverem seus *engines* polimórficos. Realizam isso através da tradução de seu código binário em uma representação temporária, onde ocorre há a reescrita do *engine*, traduzindo-a novamente para código de máquina ao fim do processo. Com isso, podemos considerá-los *malware* de análise complexa, e, consequentemente, difíceis de serem detectados (CANI et al., 2014; YOU; YIM, 2010).

Outras técnicas de alteração de código podem ser empregadas, separadamente ou em conjunto, para obter uma melhor furtividade às ferramentas de detecção. A ***Dead-Code Insertion*** – inserção de código morto – é uma técnica simples que consiste na inserção de instruções no código de um programa para mudar a sua forma – e consequentemente, uma possível assinatura – entretanto, mantendo seu comportamento. É comum o uso de instruções do tipo NOP[18] inseridas em um código na tentativa de camuflar suas intenções (YOU; YIM, 2010).

Outra técnica de ofuscação empregada em códigos maliciosos é a ***Register Reassignment*** – Reatribuição de Registradores – a cada nova geração do malware muda-se os registradores utilizados em determinadas operações, contudo, o *malware* mantém o mesmo comportamento.

[18]NOP, NOOP, No-OP, No Operation ou Sem Operação, refere-se a uma instrução assembly que não realiza efetivamente qualquer operação.

A Reordenação de Sub-Rotina – ***Subroutine Reordering*** – realiza a reordenação randômica das sub-rotinas em um código. Ao empregar esta técnica, consegue-se gerar até $n!$ variantes de código, sendo n o número de sub-rotinas. Outra transformação possivelmente aplicada é chamada de Substituição de Instrução – ***Instruction Substitution***. Ela consiste em substituir algumas instruções do código por instruções diferentes, porém, mantendo o comportamento equivalente ao da instrução original (YOU; YIM, 2010).

Com a ***Code Transposition*** – Transposição de Código – busca-se a reordenação das sequências de instruções do código original, novamente, não causando nenhum impacto em seu comportamento. Atinge-se a transposição de código de duas maneiras, uma baseada em desvios incondicionais e a outra baseada em instruções independentes. A baseada em desvios incondicionais, rearranja aleatoriamente as instruções e então recupera a ordem original de execução injetando desvios incondicionais e instruções de salto. Já a baseada em instruções independentes busca por instruções que não impactam na execução de outras instruções, reordenando-as. A segunda abordagem, apesar de complexa, é a que obtém melhores resultados (YOU; YIM, 2010).

Já a ***Code Integration*** – Integração de Código – é a técnica que consiste em realizar a união do código malicioso com o de um arquivo alvo. É uma técnica complexa e sofisticada, pois necessita que o arquivo alvo seja descompilado com sucesso para que então o código malicioso seja integrado ao código original do arquivo alvo, posteriormente recompilando-o com as alterações realizadas, fazendo assim uma nova geração do *malware*. A detecção e recuperação do arquivo original torna-se consideravelmente difícil (YOU; YIM, 2010).

3.2.2 TÉCNICAS ANTI-ENGENHARIA REVERSA

A capacidade de executar a engenharia reversa de um programa suspeito é fundamental para o entendimento de seu fluxo de execução, estrutura de código, dentre outras características. Sabendo disso, autores de *malware* implementam técnicas que visam impossibilitar ou retardar ao máximo a análise em andamento. Podemos subdividir algumas destas técnicas de defesa contra engenharia reversa em três grandes classes: ***Anti-debugging***, ***Anti-disassembly***, ***Anti-emulação***.

3.2.2.1 ANTI-DEBUGGING

Uma miríade de técnicas comumente utilizadas para evasão de análise podem ser organizadas sobre a categoria de técnicas de ***Anti-Debugging***. A partir do momento que o *malware* consegue reconhecer que está sendo executado em um *debugger*, este pode ser

capaz de alterar seu fluxo de execução normal, modificar seu código para causar uma falha ou até mesmo explorar outras vulnerabilidades conhecidas em um determinado *debugger*. O objetivo dessas técnicas é ganhar tempo: quanto mais tempo desperdiçado pelo analista, melhor (SIKORSKI; HONIG, 2012).

Malware que executam em um sistema operacional *Windows* podem utilizar, obviamente, a *Windows API* para verificar se existe um *debugger* em execução, executando algumas funções simples, tais como: ***IsDebuggerPresent*** verifica se o processo atual está sendo executado no contexto de um *debugger*; ***NtQueryInformationProcess*** verifica se um processo está sendo executado por um *debugger*; ***OutputDebugString*** é uma função utilizada para enviar uma string para ser exibida por um *debugger*; já a ***INtQueryInformationProcess*** retorna informações sobre um processo, por exemplo, passando ***ProcessDebugPort*** como um dos parâmetros da função, obtemos um valor diferente de zero caso um *debugger* esteja executando no processo consultado. Outra técnica utilizada por *malware* é o escaneamento de pontos de parada. Procurando pelo *opcode* INT 3 (0XCC), que é uma interrupção utilizada por *debuggers* para temporariamente substituir uma instrução do programa em execução para que consiga registrar um ponto de parada. *Malware* também podem eventualmente executar ***checksums*** em regiões de seu próprio código. Outro método aplicado é a verificação de tempo de execução do programa, já que um *debugger* diminui consideramente a velocidade de execução de um programa. Uma das maneiras de se realizar essa contagem de tempo é utilizando a instrução rdtsc, que retorna a contagem de *ticks* desde a última reinicialização do sistema, ou utilizando as funções ***QueryPerformanceCounter*** – acessa os registradores de contagem de atividades no processador para obter um tempo. É invocada duas vezes para se obter um delta do tempo de execução para posterior comparação; se muito tempo se passou entre as duas chamadas, assume-se que um *debugger* está em execução – e ***GetTickCound*** – esta função retorna a contagem de milisegundos decorrida desde a última reinicialização do sistema (BRANCO et al., 2012; SIKORSKI; HONIG, 2012).

Por vezes um programa malicioso pode empregar não apenas técnicas de detecção de um *debugger*, mas também, métodos de interferência e até disrupção da execução de um *debugger*. Usualmente um analista inicia o exame de um programa suspeito iniciando pelas instruções localizadas nos pontos de entrada do executável. O Ponto de Entrada é um campo localizado no cabeçalho de um arquivo no formato Windows PE que armazena o endereço da primeira instrução a ser executada em um programa. Ao executar o programa em um debugger é para esta primeira instrução que somos levados após a carga do executável. Funções de TLS callback são sempre executadas pelo sistema operacional

antes de executar o programa a partir do ponto de entrada. Desta forma, é possível executar instruções maliciosas assim que o programa for carregado no *debugger* (SIKORSKI; HONIG, 2012; ZELTSER, 2009). **Exceções** também são empregadas na detecção ou disrupção de um *debugger*. Assume-se que o debugger sempre tem o comportamento de capturar a exceção e não passá-la imediatamente para o programa. Se o *debugger* devolver a exceção capturada de maneira inadequada para o processo em execução, pode ser possível detectar esta falha através do mecanismo de manipulação de exceções do programa. Usam-se também de **Interrupções**, como a previamente mencionada INT 3, utilizada em diversos pontos do código do *malware*, enganando o *debugger* que as detecta como pontos de parada legítimos, dificultando o trabalho do analista. A INT 2D funciona de forma semelhante, porém esta interrupção é utilizada para criação de pontos de parada no *kernel*. Outra técnica é a **inserção de ICE** – *In-Circuit Emulator Breakpoint*. Essa interrupção gera uma exceção e pode levar o *debugger* a confundi-la com uma exceção comum, não executando o manipulador de exceção já estabelecido. Um *malware* pode tirar vantagem disso, usando o manipulador de exceções para executar seu fluxo de instruções, escondendo o código malicioso de um analista. Outras técnicas empregadas por autores de *malware* exploram vulnerabilidades nos *debuggers*, como por exemplo, a alteração de um cabeçalho de arquivo PE, para resultar em erro durante a sua execução por um *debugger*, apesar do código ser perfeitamente executável exteriormente (SIKORSKI; HONIG, 2012).

3.2.2.2 ANTI-DISASSEMBLY

A aplicação de mecanismos de ***Anti-Disassembly*** visam ganhar vantagem sobre as limitações dos *disassemblers* – desmontadores – e as hipóteses que devem tomar para representar o código do programa durante sua desmontagem, ou seja, exploram o comportamento do algoritmo de desmontagem e o usam como vantagem. Existem duas estratégias possíveis no que diz respeito a desmontagem de código: a Desmontagem Linear e a Desmontagem Orientada a Fluxos.

A **Desmontagem Linear**, técnica de implementação mais simples, consiste na iteração de blocos de código apenas uma instrução de cada vez. Uma característica importante desta técnica é que ela utiliza o tamanho da instrução desmontada para determinar qual *byte* deve ser desmontado em sequencia, sem se preocupar com o controle do fluxo de instruções. O grande problema com essa estratégia é que ela normalmente desmonta código desnecessariamente, algumas vezes, inclusive, de maneira incorreta, já que não se consegue diferenciar o que é uma instrução do que é um dado dentro do programa. A

outra estratégia é a **Desmontagem Orientada a Fluxos**. Um desmontador, utilizando esta técnica, analisa cada instrução encontrada e cria uma lista de locais que ainda deverão ser desmontados no futuro. Contudo, apesar de ser uma técnica mais robusta, ainda pode enfrentar problemas, principalmente ao lidar com desvios condicionais, exceções e ponteiros.

Existem ainda outros métodos de ***Anti-Disassembly*** mais básicos (SIKORSKI; HONIG, 2012; BRANCO et al., 2012): ***Garbage Bytes*** é um método que consiste na adição de *bytes* adicionais após uma instrução de maneira a confundir o desmontador que esses *bytes* adicionais fazem parte de alguma outra instrução, gerando assim código desmontado inválido. Uma outra técnica consiste na **Mudança Incondicional no Fluxo de Controle do Programa**, deixando alguma outra área de código contendo outra técnica *anti-disassembly* inalcançável durante a execução. Um exemplo desta técnica é a utilização de instruções JMP incondicionais em associação a técnica *Garbage Bytes*. Outra técnica utilizada consiste na utilização de **Saltos Condicionais Falsos**, geralmente em associação com outras técnicas *anti-disassembly*, leva o desmontador a desmontar código que normalmente não seria executado, gerando uma interpretação errada do código do programa analisado. O ***Call Trick*** consiste na alteração do endereço de retorno da função, fazendo com o que o desmontador confunda o final do corpo da função. Já o **Redirecionamento de Fluxo no Meio de uma Instrução**, consiste, por exemplo, em esconder uma instrução no meio de uma outra instrução, fazendo com que o desmontador mostre uma instrução que não é executada em tempo de execução, ao invés da instrução correta, que reside escondida no meio das outras.

3.2.2.3 ANTI-EMULAÇÃO

Ambientes emulados são uma ameaça aos *malware*, já que esse é o único meio de se analisar um programa suspeito durante o tempo de execução. As técnicas ***Anti-emulação*** *ou* ***Anti-Virtual Machines*** tem como objetivo frustar tentativas de análise em ambientes virtuais, como máquinas virtuais ou *sandboxes*. Ao ser executado em uma máquina virtual, o *malware* detecta o ambiente e camufla seu comportamento padrão, assumindo um comportamento de um programa inócuo ou até mesmo interrompendo sua execução. Essas técnicas são usualmente empregadas em *malware* largamente distribuídos, como *bots* e *spywares*, já que possuem como alvos máquinas de usuários finais, que comumente não rodam máquinas virtuais. Contudo, esse tipo de prática vem caindo em desuso devido à utilização cada vez mais abrangente de máquinas na nuvem. Esses ambientes rodam

em máquinas virtualizadas, não indicando portanto que uma plataforma de análise de malware está sendo utilizada (SIKORSKI; HONIG, 2012). Métodos *Anti-VM* geralmente utilizam como alvo o *VMWare*, que é uma plataforma de virtualização muito difundida. Por isso, as técnicas descritas nesta seção utilizam como foco vulnerabilidades ou características deste ambiente.

O uso de determinadas **Instruções de CPU**, quando executadas em ambientes virtualizados, possuem resultados característicos e indicam que um ambiente virtualizado esteja em execução. É o caso das instruções *SIDT – Store Interrupt Descriptor Table Register*, *SLDT – Store Local Descriptor Table Register*, *SGDT – Store Global Descriptor Table Register*, *STR – Store Task Register* e *SMSW – Store Machine Status Word*.

Artefatos do *VMWare* podem ser utilizados para identificação de execução em um ambiente virtual. Muitos destes artefatos estão presentes no sistema de arquivos, no registro do sistema operacional e na listagem de processos em execução. Procurando por *VMWare* na listagem de processos, pode-se encontrar os processos *VMwareService.exe*, *VMwareTray.exe* e *VMwareUser.exe*. A mesma busca executada no registro de uma máquina virtual *VMWare* nos leva a várias chaves de registro, informando sobre o disco virtual, mouse e adaptadores. O mesmo se repete para a busca em memória física, trazendo artefatos que indicam a execução no ambiente virtualizado.

3.3 ANÁLISE DINÂMICA

A análise dinâmica é o processo de análise do comportamento de um software durante o seu tempo de execução, consistindo no monitoramento da interação entre o programa analisado e o sistema operacional – sistema de arquivos, processos em execução, rede, memória, etc. – e os efeitos de sua execução. O programa suspeito é frequentemente executado em um ambiente controlado, como por exemplo, uma máquina virtual, um emulador, um simulador ou uma *sandbox* (BAYER et al., 2006; SIKORSKI; HONIG, 2012).

Em contraposição à análise estática, a dinâmica provê meios de detecção de *malware* desconhecidos, sendo imune a técnicas de ofuscação. Contudo, programadores dessas ameaças desenvolvem-nas com capacidade para detectar se estão sendo executadas em um ambiente de emulação ou máquina virtual, fazendo com que executem de maneira distinta – como um programa inócuo – ou que nem mesmo cheguem a executar.

Para a realização da análise dinâmica, um analista poderá aplicar uma série de abordagens e ferramentas para auxilia-lo no trabalho, como por exemplo (EGELE et al., 2008):

Monitoramento de Chamadas de Função: que é o ato de monitorar quais funções são chamadas pelo programa, normalmente realizada através da interceptação[19], ato de interceptar uma chamada de função, manipulando o programa analisado para que este invoque uma outra função – uma **função *hook*** – que contenha a capacidade para realizar a análise, seja para analisar os parâmetros de entrada ou para realizar o armazenamento das invocações em um arquivo de *log* dessas chamadas;

Análise de Parâmetros de Função: Ao contrário de sua contraparte da análise estática que tenta por inferência obter o conjunto de valores possíveis para os parâmetros, a dinâmica foca nos valores atuais dos parâmetros passados quando a função foi invocada.

Rastreamento do Fluxo de Informações[20]: É a análise de como um programa processa dados durante sua execução. Em geral, os dados interessantes para a análise são marcados com um rótulo e sempre que o dado marcado for processado pela aplicação ele então é propagado.

Rastreamento de Instruções – ***instruction trace***. Uma fonte valiosa de informação para avaliar o comportamento de um programa, pois deve conter informações importantes não representadas em um alto nível de abstração, como por exemplo, em relatórios de chamadas de funções.

3.4 ANÁLISE ESTÁTICA

A análise estática é o processo de analisar o código ou estrutura do arquivo executável suspeito sem a necessidade de executá-lo. Realiza-se através da extração de informações de baixo nível diretamente do arquivo, seja através de descompilação ou desmontagem do código. A desvantagem da aplicação da análise estática é que esta pode falhar na análise de *malware* cujos padrões de estrutura de código sejam desconhecidos, principalmente os que se utilizam de técnicas de ofuscação de código. As grandes vantagens são o seu tempo de execução, que é rápido e é uma técnica segura já que não realiza a execução do arquivo suspeito, ao contrário da análise dinâmica, que demanda muito tempo de análise tanto para analisar a execução de vários cenários, como de montagem do ambiente controlado onde o arquivo será analisado. Uma outra vantagem da análise estática é que se consegue analisar *malware multi-path*, ou seja, averiguar múltiplos ramos ou desvios de execução no código (MATHUR; HIRANWAL, 2013; SIKORSKI; HONIG, 2012).

A análise estática pode ser dividida em Análise Básica e Avançada. Na análise bá-

[19] ***Hooking***
[20] ***Taint Analysis***

sica, o analista de segurança emprega ferramentas para uma análise inicial do executável. Empregan-se diversos **antivírus** contra o arquivo suspeito, realizando a comparação de ***hashes***[21] e extração de outras informações diretamente do arquivo: buscando por **Strings**, atrás de mensagens de erro, *URLs*, nomes de funções chamadas pelo arquivo, importações de bibliotecas (*DLLs*) e outras informações disponíveis no cabeçalho do arquivo (SIKORSKI; HONIG, 2012). Na análise estática avançada, trabalha-se com engenharia reversa, analisando o código desmontado do programa procurando entender os objetivos do programa analisado (SIKORSKI; HONIG, 2012; MANGIALARDO, 2015).

[21] ***Hashing***, que é uma técnica simples para identificar unicamente um arquivo. Extraindo uma *hash* de um *malware*, pode-se então comparar esta assinatura contra o *hash* de outros arquivos suspeitos. Alguns algoritmos de *hashing* frequentemente utilizados são o MD5 e o SHA-1.

4 TRABALHOS RELACIONADOS

Como já mencionado anteriormente, a detecção e classificação de malware são trabalhos complexos e super especializados. Comumente requer o emprego de profissionais especialistas e o uso de um conjunto ferramentas adequadas. Muitos programas maliciosos possuem a capacidade de se camuflar ou alterar seu próprio conteúdo com o intuito de escapar da detecção. Além disso, malware *zero-day*[22] surgem a todo momento, tornando boa parte das ferramentas baseadas em assinaturas de arquivo ou heurísticas incapazes de realizar o trabalho.

Procurando soluções de detecção e classificação resilientes a essas técnicas de evasão, podemos observar uma grande quantidade de trabalhos científicos abordando o problema utilizando a mineração de dados e algoritmos de aprendizado de máquina, seja aplicado a analise estática ou dinâmica.

Os trabalhos aqui expostos podem ser divididos em três categorias, sendo elas: i) **Trabalhos Relacionados ao Aprendizado Profundo**, onde sumarizamos trabalhos relacionados diretamente ao Aprendizado Profundo, explorando técnicas e arquiteturas de redes profundas, ii) **Trabalhos Relacionados à Análise de *Malware***, onde descrevemos trabalhos focados na análise de *malware*, estática, dinâmica ou mista, utilizando aprendizado de máquina como parte do método, e iii) **Aprendizado Profundo como Ferramenta para a Análise de *Malware***, onde a utilização do Aprendizado Profundo faz parte da abordagem ao problema de classificação ou detecção de *malware*.

4.1 TRABALHOS RELACIONADOS COM APRENDIZADO PROFUNDO

Redes neurais artificiais com múltiplas camadas ocultas de processamento permitem que representações de dados sejam aprendidas em múltiplos níveis de abstração, criando assim uma Hierarquia de Conceitos. Diferentemente de algoritmos de aprendizado de máquina tradicionais, redes neurais de Aprendizagem Profunda não necessitam que um especialista em um domínio realize o tratamento e transformação dos dados brutos e nem a escolha das características relevantes para a aplicação da técnica. As arquiteturas desses modelos, técnicas de treinamento e a importância e vantagens do aprendizado de representações é

[22] Malware que explora uma vulnerabilidade de segurança no mesmo dia em que ela se tornou conhecida, antes mesmo que uma correção possa ser disponibilizada (Fonte: https://www.kaspersky.com.br/resource-center/definitions/zero-day-exploit).

o foco dos trabalhos à seguir.

"Representation Learning: A Review and New Perspectives"(BENGIO et al., 2013), os autores desenvolvem o tema Aprendizado de Representações focando especificamente nos métodos de Aprendizado Profundo. Este estudo detalha características que influenciam na produção de uma boa representação, como organização de conceitos em forma hierárquica, Aprendizado Semi-supervisionado, esparsidade, coerência espacial e temporal, dentre outros. Detalha-se também alguns modelos de Aprendizado de Representação, como redes autocodificadoras e outras técnicas.

Em **"Why does unsupervised pre-training help deep learning?"**(ERHAN et al., 2010) abordam-se questões relacionadas a robustez e a eficiência trazida aos modelos de Aprendizado Profundo pelo pré-treinamento não supervisionado. Este estudo demonstra através de uma série de experimentos a influência do pré-treinamento em relação à profundidade da arquitetura, capacidade do modelo e o número de exemplos de treinamento, bem como o efeito regularizador não usual que este tipo de treinamento traz para arquiteturas profundas. Para os experimentos foram considerados dois modelos de arquiteturas profundas, a *Deep Belief Networks* (DBN) e o *Stacked Denoising Autoencoders* (SDAE) e utilizaram-se de três bases de dados: O *MNIST*, *InfiniteMNIST* e o *Shapeset*.

4.2 TRABALHOS RELACIONADOS COM ANÁLISE DE MALWARE

Em **"Detecting unknown malicious code by applying classification techniques on OpCode patterns"**(SHABTAI et al., 2012), os autores tem como objetivo explorar métodos utilizando técnicas de mineração de dados para criar modelos com boa performance na detecção de arquivos binários desconhecidos, ou seja, ainda não observados pelo classificador. Com esse objetivo, realizaram um estudo aprofundado sobre a utilização de padrões de n-gramas de OpCodes e sua capacidade como boa representação para classificação de *malware* desconhecido. Em sua pesquisa, utilizam-se de representações de termo (TF e TF-IDF), diversos n-gramas (de 1 a 6-gramas) e técnicas de seleção de características (*Document Frequency* - DF, *Document Frequency* - DF, *Fisher Score* - FS e *Gain Ratio* - GR). Também realizaram testes com diversos classificadores, simulando condições da vida real, como o desbalanceamento de classes benigna e maligna, e avaliaram com qual frequência os classificadores deveriam ser retreinados com amostras desconhecidas para que este aumente em performance. Dentre os classificadores utilizados o que mais se destacou em performance foi o *Random Forest* que obteve aproximadamente 95.1% de

acurácia.

O artigo **"Integrating Static and Dynamic Malware Analysis Using Machine Learning"**(MANGIALARDO; DUARTE, 2015) apresenta um método para construção de uma base de dados, unificando características extraídas através de análise estática e dinâmica, para a análise e classificação automática de *malware* em diversos tipos. Dois algoritmos de árvores de decisão foram utilizados, o C5.0 e o *Random Forest*. Nove experimentos foram executados, sendo três realizando a comparação dos dois algoritmos por desempenho na identificação de *malware* utilizando classificação binária e os restantes considerando o problema de categorizar os *malware* por tipo ou multi-classificação. Como resultado dos experimentos, verificou-se que o algoritmo que obteve o melhor desempenho foi o *Random Forest*. A acurácia da análise para o problema de classificação binária foi igual a 95,75% e, para o problema de multi-classificação, foi igual a 93,02%. Ainda foi possível observar que em todos os experimentos conduzidos o desempenho da análise unificada, considerando tanto características de provenientes de análise estática como dinâmica, foi superior a execução dos experimentos com características das análises isoladas.

Em **"DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket"**(ARP et al., 2014), foi desenvolvido um método de análise estática para identificação de *Malware* em aplicações *Android*. Durante a análise de um arquivo no formato *Android Application Package* (APK), características foram extraídas e organizadas em conjuntos de *strings*, como por exemplo, permissões de usuário, chamadas à API, endereços de rede, dentre outros; e incorporados em um espaço vetorial conjunto para que finalmente fossem classificadas por um algoritmo de aprendizado de máquina, no caso a *Support Vector Machines* (SVM). A grande vantagem do método é a capacidade de prover uma justificativa para a detecção realizada à partir dos padrões de características indicativas de um determinado *malware*. Outra vantagem é que o método empregado é eficiente e leve, permitindo que seja executado no próprio *smartphone*. O modelo, porém, apresenta limitações inerentes a análise estática, como por exemplo, não conseguir detectar alguns métodos de evasão implementados por *malware*, como ofuscação e encriptação de código. Por utilizar-se de um modelo de aprendizado de máquina, DREBIN necessita de uma boa representatividade nos dados, tanto em amostras benignas quanto em amostras identificadas como maliciosas, sendo esta a mais difícil de se obter. Além disso, ele pode degradar a performance de um classificador utilizando-se de mimetismo ou ataques de envenenamento. Apesar das limitações, o modelo desenvolvido conseguiu eficientemente detectar *malware* com um baixo número de falsos positivos.

"Opcode sequences as representation of executables for data-mining-based unknown malware detection"(SANTOS et al., 2013) emprega técnicas de mineração de dados e aprendizado de máquina utilizando as frequências de sequências de *opcodes* obtidas à partir de uma base de *malwares* com arquivos no formato *Portable Executable* (PE) extraídas do *website VxHeavens*. Diversos classificadores foram utilizados com a hipótese de que seria possível identificar *malware* desconhecido à partir da representação gerada. Os melhores resultados foram obtidos utilizando *Support Vector Machines* (SVM) com *kernel* polinomial e Árvore de Decisão, que mantiveram uma alta acurácia com baixo número de falsos positivos. Apesar da hipótese ter sido comprovada com os resultados dos experimentos, ainda existem algumas considerações sobre a viabilidade do método, como por exemplo, o custo de processamento das sequências de *opcode* ser altamente dependente do tamanho das sequências geradas, a utilização apenas de *opcodes* e não dos operadores, e devido a natureza estática da técnica, a impossibilidade de detectar *malware* empacotado.

4.3 APRENDIZADO PROFUNDO COMO FERRAMENTA PARA A ANÁLISE DE MALWARE

Em **"Malware detection based on deep learning algorithm"** (YUXIN; SIYI, 2017), os autores abordaram o problema da detecção de malware utilizando *Deep Belief Networks*, procurando averiguar sua capacidade como base para um classificador e também sua capacidade como extratora de características. O trabalho utilizou-se de amostras de três conjuntos de malwares diferentes – Microsoft Malware Dataset, VxHeavens e offensive-computing.net – além de um outro conjunto apenas com programas benignos. O conjunto de dados final apresenta a mesma distribuição entre as classes Benigna e Maligna. Foram utilizados procedimentos de extração de sequências de opcodes dos arquivos binários, gerando vetores de trigramas baseados nessas sequências de opcode. Para diminuir o tamanho dos vetores, vetores com os 100,200,300 e 400 melhores trigramas foram utilizados, baseados em métodos de seleção de características, tais como a Frequência do Documento (DF) e o Ganho de Informação. As DBNs foram treinadas por meio do algoritmo de Pré-treinamento por camadas utilizando *Restricted Boltzmann Machines*, onde, posteriormente, é treinada em conjunto com a camada de classificação. Para estabelecer uma linha de base de comparação, utilizaram outros três classificadores: Support Vector Machines (SVM), Árvores de Decisão (DT) e *K-nearest neighbor* (KNN). Em comparação com um trabalho relacionado que utiliza SVM, DT e KNN, os modelos gerados pelas

DBNs obtiveram melhor performance, sendo sua acurácia média superior de 1 a 2% e o *F1 Score*, de 0.01 a 0.02 melhor em relação ao melhor classificador base (DT). Foi verificado também que DBNs conseguiram extrair boas características. Para isso, a DBN foi treinada como uma rede autocodificadora, utilizando tanto um conjunto de dados rotulado e um conjunto misto, onde existem amostras rotuladas e não rotuladas, extraindo os vetores de 50, 100, 150 e 200 características diretamente da camada oculta da rede. Com esses vetores de características, foram treinados então três tipos de classificadores, SVM, DT e KNN. De acordo com os autores, os classificadores treinados com os vetores extraídos de DBNs treinadas no conjunto de dados misto (rotulados e não rotulados) obtiveram uma performance levemente superior aos dos treinados apenas com os vetores extraídos de DBNs treinadas apenas nos dados rotulados.

"Static Analysis on Disassembled Files: A Deep Learning Approach to Malware Classification" (PINTO; DUARTE, 2017) foi o artigo onde foram relatados os primeiros experimentos com o método descrito no capítulo 5 desta dissertação. A abordagem proposta é o treinamento de redes autocodificadoras profundas – treinamento não-supervisionado – com a intenção de capturar as melhores características advindas do que chamamos de Bag of OPCODEs, Opcodes e seus operandos no formato de Bytes em hexadecimal que são extraídos do *malware* no formato Windows PE desmontados – sem qualquer pré-seleção baseada em conhecimento prévio das famílias de malware – produzindo vetores para unigramas, bigramas e trigramas, posteriormente ponderados pelo TF-IDF. Após o treinamento da autocodificadora, iniciamos a fase de treinamento supervisionado, onde pré-inicializamos uma Multilayer Perceptron (MLP) com os pesos aprendidos pelas redes autocodificadoras treinadas anteriormente. O conjunto de dados utilizado foi um subconjunto do Microsoft Malware Challenge Dataset, contendo amostras das duas classes mais representativas e as amostras da classe menos frequente no conjunto de dados. Tanto o conjunto de dados original quanto o subconjunto utilizado não possui uma distribuição de amostras balanceada, o que geralmente é problemático em problemas de classificação ou detecção. Três modelos foram gerados: dois para vetores de unigramas e um para vetores de bigramas. As autocodificadoras treinadas foram sempre subcompletas – onde as camadas internas da rede possuem dimensionalidade menor que as camadas de entrada e saída. O melhor resultado obtido por um modelo apresentou uma acurácia média de 99,55%, o que indica o potencial de utilização da metodologia apresentada.

Em **"DL4MD: A Deep Learning Framework for Intelligent Malware Detection"** (HARDY et al., 2016), os autores projetaram um *framework* com Aprendizado Profundo para detecção de *malware* em arquivos no formato *Portable Executable* (PE) do

sistema operacional *Microsoft Windows*. O *framework* é composto de dois componentes: O primeiro, um Extrator de Características, responsável por descomprimir o arquivo PE e pela extração das chamadas a *Windows Application Programming Interface* (*Windows* API) e sua conversão em assinaturas de 32-bits; O segundo, um Classificador baseado em Aprendizado Profundo, utilizando *Stacked Autoencoders*, treinado com a técnica de pré-treinamento não supervisionado. Diversas arquiteturas do modelo proposto foram experimentadas, diversificando a quantidade de camadas ocultas da rede e quantidade de neurônios por camada. Os experimentos também compararam o *framework* em relação aos métodos tradicionais de aprendizado de máquina - Redes Neurais Artificiais, *Support Vector Machines* (SVM), *Naïve Bayes* e Árvore de Decisão. A arquitetura que alcançou a melhor performance era constituída de três camadas ocultas com cem neurônios cada. De acordo com os autores, o *framework* alcançou uma performance superior aos métodos tradicionais de aprendizado de máquina, mostrando-se viável para detecção de *malware*.

"Droid-Sec: Deep Learning in Android Malware Detection" (YUAN et al., 2014) é uma técnica híbrida para detecção de *malware*, utilizando características extraídas à partir de análise estática e dinâmica em aplicações *Android* e utilizando uma *Deep Belief Network* (DBN). O conjunto de dados foi composto de 250 *malwares* obtidas no *contagio mobile* e para a classe de aplicações normais, 250 aplicações advindas da *Google Play Store*. Portanto, os conjuntos de treinamento e de teste possuem uma distribuição igual entre as duas classes. Foram experimentadas diversas configurações da arquitetura da DBN, variando o número de camadas ocultas e o número de neurônios. A arquitetura que obteve a melhor acurácia era composta de três camadas ocultas de 150 neurônios cada. O modelo desenvolvido também foi testado contra métodos tradicionais de aprendizado de máquina – *Support Vector Machines*, C4.5, *Naïve Bayes*, Regressão Linear e *Multi-layer Perceptron* – obtendo a melhor acurácia dentre os testados.

"Large-scale malware classification using random projections and neural networks" (DAHL et al., 2013) desenvolveu um sistema para classificação de *malware* em larga escala. O conjunto de dados é composto por 2.6 milhões de arquivos, sendo 18.433.593 malignos e 817.485 benignos. Foi aplicada a técnica chamada *Random Projections* para realizar uma redução de dimensionalidade, diminuindo de 179 mil características para 4 mil. Os classificadores escolhidos para os experimentos foram a Regressão Logística e a *Deep Belief Networks* (DBN). Verificaram que redes neurais treinadas a partir das projeções obtiveram 43% de redução da taxa de erro. Os experimentos procuraram testar diversas configurações, como por exemplo a quantidade de camadas ocultas em uma rede, quantidade de neurônios por camada e a quantidade de projeções a serem

utilizadas como entrada para a rede. O melhor resultado foi alcançado por um comitê composto por 9 redes neurais, com uma taxa de falsos positivos de 0.01%.

4.4 COMPARATIVO SOBRE OS TRABALHOS APRESENTADOS

Os trabalhos listados anteriormente serviram de inspiração em diversas partes neste trabalho. Podemos observar diversas características, semelhanças e diferenças entre esses trabalhos nas tabelas 4.1 e 4.2.

(SHABTAI et al., 2012; SANTOS et al., 2013) exploraram a utilização de opcodes como características facilitadoras para detecção de *malware*. Neste ponto, não só inspiraram a utilização dos opcodes em nosso modelo, como também nos ajudaram a incluir os operandos dos opcodes – seus parâmetros, como possíveis características. Podemos observar que nenhum dos trabalhos que utilizaram opcodes também utilizaram os operandos extraídos dos programas desmontados (SHABTAI et al., 2012; SANTOS et al., 2013; YUXIN; SIYI, 2017). (SHABTAI et al., 2012) também contribuiu para a definição de um mecanismo de extração/seleção de características utilizado, o TF-IDF. A utilização de combinações de opcodes foram testadas em (SHABTAI et al., 2012; SANTOS et al., 2013; DAHL et al., 2013), com resultados diversos quanto ao número de n-gramas. Pareceu-nos razoável explorar essas combinações, o que foi atestado com relativo sucesso em (PINTO; DUARTE, 2017). A escolha por iniciarmos os testes das arquiteturas de aprendizado profundo pelas redes autocodificadoras veio através dos bons resultados obtidos em (HARDY et al., 2016) e com um embasamento teórico para o uso do pré-treinamento não-supervisionado para posterior treinamento de um classificador em (BENGIO et al., 2013; ERHAN et al., 2010). (YUXIN; SIYI, 2017) foi incluído por possuir semelhanças no processo de treinamento e classificação e por também utilizar o mesmo conjunto de dados escolhido para esta dissertação. Ainda assim, apesar do nosso método também realizar pré-treinamento não-supervisionado, ele se distingue, já que não utilizamos o esquema de treinamento guloso camada-a-camada implementado tanto em (HARDY et al., 2016) quanto em (YUXIN; SIYI, 2017), realizamos o treinamento da rede autocodificadora por completo. Além disso, nossos experimentos iniciais (PINTO; DUARTE, 2017) nos mostraram que devíamos explorar outras configurações de arquiteturas de autocodificadoras, tanto em quantidade de camadas quanto no número de neurônios por camada, o que nos levou a realizar uma análise de sensibilidade nos modelos.

TAB. 4.1: Tabela comparativa de características dos trabalhos relacionados

Referência	**OpCodes**	**Operandos**	**N-Gramas**	**TF-IDF**	**AE**	**AD**	**AM**	**AP**
Shabtai et al. (2012)	✓	×	✓	✓	✓	×	✓	×
Santos et al. (2013)	✓	×	✓	×	✓	×	✓	×
Dahl et al. (2013)	×	×	✓	×	✓	✓	✓	✓
Yuan et al. (2014)	×	×	×	×	✓	✓	✓	✓
Arp et al. (2014)	×	×	×	×	✓	×	✓	×
Mangialardo e Duarte (2015)	×	×	×	×	✓	✓	✓	×
Hardy et al. (2016)	×	×	×	×	✓	×	✓	✓
Yuxin e Siyi (2017)	✓	×	✓	×	✓	×	✓	✓
Pinto e Duarte (2017)	✓	✓	✓	✓	✓	×	×	✓
Esta dissertação	✓	✓	✓	✓	✓	×	×	✓

TAB. 4.2: Tabela comparativa Conjunto de Dados e Plataformas

Trabalho	**Conjunto de Dados**	**Plataforma**
Shabtai et al. (2012)	VxHeaven e Próprio (Benigno)	Windows
Santos et al. (2013)	VxHeaven e Próprio (Benigno)	Windows
Dahl et al. (2013)	Próprio	Windows
Yuan et al. (2014)	Contagio Mobile	Android
Arp et al. (2014)	Android Genome Project e outros	Android
Mangialardo e Duarte (2015)	Próprio	Windows
Hardy et al. (2016)	Comodo Cloud Security Center	Windows
Yuxin e Siyi (2017)	Microsoft Malware Classification Challenge, VxHeaven, OffensiveComputing	Windows
Pinto e Duarte (2017)	Microsoft Malware Classification Challenge	Windows
Esta dissertação	**Microsoft Malware Classification Challenge**	**Windows**

5 UMA ABORDAGEM BASEADA EM APRENDIZADO PROFUNDO PARA A ANÁLISE DE ESTÁTICA DE MALWARE

Neste capítulo detalharemos nossa solução proposta para o problema de classificação de *malware*, um problema complexo, porém essencial para análise de malware, que é a correta identificação do código malicioso em suas famílias. Em vários trabalhos anteriores (BILAR, 2007; SHABTAI et al., 2012; SANTOS et al., 2013) investigou-se a utilização de OpCodes como características determinantes para a identificação correta de código malicioso, porém, como já indicado previamente, nenhum dos trabalhos que tivemos conhecimento abordavam o uso do OpCode juntamente com seus operandos. Outrossim, este trabalho é centrado na investigação do emprego de redes neurais de aprendizado profundo para este tipo de problema, onde procura-se analisar o impacto que o ajustes dos parâmetros de tais redes pode causar no desempenho do modelo treinado. Tudo isso sem a utilização de conhecimento de domínio ou qualquer outro conhecimento advindo das famílias de *malware* investigadas neste estudo. Dito isso, podemos dividir a solução proposta em duas partes: i) **Construção do conjunto de dados**, onde ocorre a extração dos dados relevantes diretamente do código desmontado, sua transformação, ponderação e normalização, para que então se inicie a fase de ii) **Treinamento e Validação de Modelos**, onde redes são treinadas tanto para a descoberta automática de padrões de códigos maliciosos, procurando aprender uma espécie de "assinatura" do *malware*, quanto para a classificação dos mesmos dentre as diversas famílias estudadas.

5.1 CONSTRUÇÃO DO CONJUNTO DE DADOS

Nesta seção descrevemos as técnicas aplicadas para alteração da representação de um conjunto de dados padrão para a formação do conjunto de dados experimental deste trabalho.

5.1.1 O MODELO SACO DE PALAVRAS

O Saco de Palavras é um modelo de representação de linguagem [23] onde as frequências dos termos ou palavras de um documento são computadas e representadas na forma de um vetor, onde cada posição do vetor representa um termo ou palavra que foi encontrado

[23] Também é conhecido como *Bag of Words*, modelo espaço vetorial ou *Vector Space Model*

no documento e o valor correspondente representa a frequência do termo. Apesar de utilizado com sucesso, uma desvantagem deste modelo é que se perde qualquer relação de ordem entre as palavras, bem como regras gramaticais, propriedades essas que poderiam servir para melhor distinguir um documento do outro (MANNING et al., 2008b). Um exemplo descrito por Manning et al. (2008b), está nos documentos com o conteúdo *"Maria é mais veloz que João"* e *"João é mais veloz que Maria"*, onde pela representação criada pelo modelo de Saco de Palavras torna esses documentos idênticos.

5.1.2 TF-IDF: *TERM FREQUENCY – INVERSE DOCUMENT FREQUENCY*

O *Term Frequency - Inverse Document Frequency* (TF-IDF) é uma técnica de ponderação que une duas outras técnicas: A *Term Frequency*, frequência do termo, e a *Inverse Document Frequency* que é a inversa da frequência do documento.

A frequência do termo $tf_{t,d}$ representa em sua forma mais corriqueira a contagem do termo t no documento d, ou seja, a quantidade de vezes em que uma palavra aparece em um documento. No entanto, utilizando o tf, considera-se que todos os termos são igualmente importantes em relação a sua relevância. Para atenuar situações em que um termo possui sempre uma frequência muito alta, utiliza-se a ponderação pelo IDF, que é representado por

$$idf_t = log\frac{N}{df_t} \tag{5.1}$$

onde N é o número total de documentos em uma coleção e df_t é a quantidade de documentos em que o termo t aparece. O peso calculado de uma palavra rara é alto, enquanto o peso calculado para uma palavra muito frequente é baixo. Dadas as definições, chegamos então ao $tf - idf$, que atribui um peso para o termo t em um documento d o que pode ser definido como

$$\text{tf-idf}_{\text{t,d}} = tf_{t,d} \times idf_t \tag{5.2}$$

onde podemos afirmar que: para um termo t que ocorre muitas vezes em um número pequeno de documentos, o mecanismo atribui um peso bem alto. Um peso baixo pode ser atribuído a t caso ocorra poucas vezes em um documento ou ocorra em muitos documentos (pouca relevância). O peso mais baixo é destinado ao t que ocorre em praticamente todos os documentos (MANNING et al., 2008b).

Existem várias fórmulas para se calcular o TF-IDF. A implementação que é utilizada neste trabalho é o *TfIdfVectorizer* da biblioteca *Scikit-Learn* (PEDREGOSA et al., 2011).

5.1.3 CONJUNTO DE DADOS DE REFERÊNCIA

Em 2015, a Microsoft patrocinou um campeonato chamado *Microsoft Malware Classification Challenge (BIG 2015)*, cujo objetivo foi o de realizar a classificação de *malwares* dentre diversas famílias. Para isso, um conjunto de dados com amostras de malwares de nove famílias foi disponibilizado (Tabela 5.1) (MICROSOFT, 2015; RONEN et al., 2018).

TAB. 5.1: Distribuição de amostras através das famílias de malware no conjunto de treinamento (MICROSOFT, 2015; RONEN et al., 2018)

Classe	**Família**	# **Amostras**
1	Ramnit	1541
2	Lollipop	2478
3	Kelihos_ver3	2942
4	Vundo	475
5	Simda	42
6	Tracur	751
7	Kelihos_ver1	398
8	Obfuscator.ACY	1228
9	Gatak	1013
-	**Total**	**10868**

O conjunto é composto por amostras separadas para treinamento, onde todas as amostras são identificáveis, e testes, que contém amostras não rotuladas. Para cada amostra do conjunto de dados existem dois arquivos: Um arquivo com a extensão '.byte' que contém uma representação hexadecimal do arquivo executável do malware, excluindo seu cabeçalho, e um arquivo com a extensão '.asm', contendo o código desmontado e outras informações, como comentários, chamadas de funções, dentre outras, extraídas do arquivo executável por um programa de desmontagem (HEX-RAYS, 2017). Durante a execução deste trabalho, para a tarefa de classificação de malware, adotou-se as amostras do conjunto de treinamento, ou seja, as amostras rotuladas do conjunto descrito, como o conjunto de dados padrão para os experimentos.

5.1.4 EXTRAÇÃO DOS OPCODES E OPERANDOS

Podemos considerar que o conjunto de dados originalmente usado é um conjunto $A = \{a_1, a_2, a_3, \cdots, a_n\}$, onde a_n é um arquivo de malware desmontado e sua cardinalidade $|A| = 10868$ arquivos.

Sendo assim, na **primeira etapa** da preparação dos conjuntos de dados, para todo $a \in A$, extraímos os opcodes e seus operandos (Figura 5.1) considerando apenas termos

```
.text:00419BA4 8B 5D 08          mov     ebx, [ebp+arg_0]
.text:00419BA7 8D 5C 9D E0       lea     ebx, [ebp+ebx*4+var_20]
.text:00419BAB 8B 33             mov     esi, [ebx]
.text:00419BAD 8B CE             mov     ecx, esi
.text:00419BAF 23 CF             and     ecx, edi
```

FIG. 5.1: Excerto de uma amostra de arquivo '.asm' do conjunto de dados utilizado. Em negrito, destacam-se os OpCodes e os operandos em formato de Bytes em hexadecimal.

que sejam bytes na base hexadecimal, excluindo qualquer dado que divirja deste tipo – dados sobre as seções do arquivo *Windows PE*, endereçamento de memória, comentários, chamadas a funções da API, dentre outros – gravando as sequências de instruções e operandos, na ordem de sua ocorrência, em um novo arquivo $i \in I$, onde inicialmente $I = \emptyset$. Temos que ao final desta etapa o conjunto de todos os arquivos advindos desta extração, $I = \{i_1, i_2, i_3, \cdots, i_n\}$ e $|I| = |A|$.

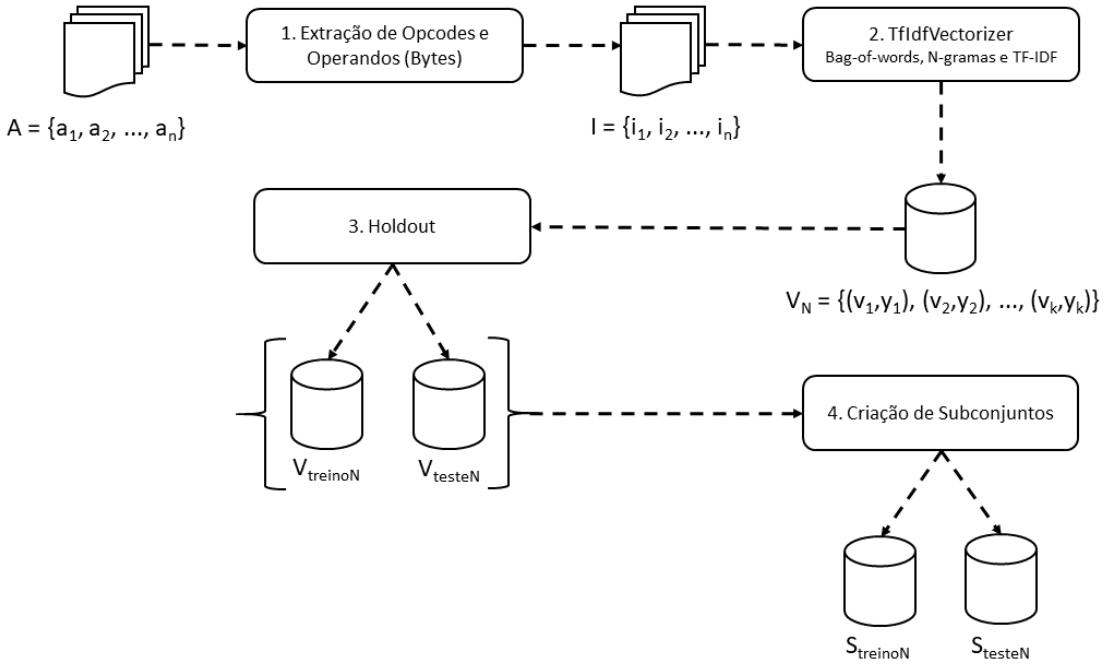

FIG. 5.2: Da extração de opcodes e operandos até o conjunto de dados de vetores para N-gramas já preparados para treinamento e validação dos modelos.

Com a formação do conjunto I, prosseguimos para a **segunda etapa** da preparação dos conjuntos de dados. Para isso, utilizamos como implementação de referência o *TfIdfVectorizer*[24] da biblioteca *Scikit Learn* (PEDREGOSA et al., 2011). Inicialmente o

[24] `http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html`

conjunto $V_N = \emptyset$, onde N indica o tipo dos vetores de saída da extração de n-gramas: por exemplo, quando $N = 1$, V_1 representa o conjunto de vetores extraídos para unigramas [25]. Para todo $i \in I$, executa-se o procedimento de tokenização e contagem de cada termo [26], e ponderação pelo *TF-IDF*, produzindo assim um vetor para cada i. Neste momento, o vetor de características é associado ao seu rótulo correspondente, um vetor no formato ***one-hot encoding***[27] tornando-se uma tupla (v_k, y_k); sendo assim, obtemos ao final desta etapa o conjunto $V_N = \{(v_1, y_1), (v_2, y_2), (v_3, y_3), \cdots, (v_k, y_k)\}$ e $|V| = |I|$.

Na **terceira etapa**, realizamos a separação de amostras para os conjuntos V_N em dois subconjuntos distintos, um para treinamento e outro para testes (Tabela 5.1.4), respectivamente V_{treino_N} e V_{teste_N}, obedecendo como critério de separação das amostras, para cada classe de malware, 75% de amostras para o conjunto de treinamento e 25% para o conjunto de testes, retiradas aleatoriamente do conjunto de dados de treinamento do conjunto de dados de referência, porém respeitando a distribuição de amostras por classe. Outros dois conjuntos oriundos de V_{treino_N} e V_{teste_N} foram gerados com o objetivo de realizar testes no treinamento dos modelos de redes neurais, respectivamente S_{treino_N} e S_{teste_N}, obedecendo os mesmos critérios de amostragem realizados anteriormente, sendo seu único diferencial o limite da cardinalidade destes conjuntos serem menores do que a cardinalidade de seus conjuntos originários, ou seja, $|S_{treino_N}| < |V_{treino_N}|$ e $|S_{teste_N}| < |V_{teste_N}|$.

TAB. 5.2: Distribuição de amostras do conjunto de dados original em um conjunto de treinamento e um conjunto de validação.

Classe	**Família**	**# Amostras**		
		Total	**Treinamento**	**Validação**
1	Ramnit	1541	1155	386
2	Lollipop	2478	1858	620
3	Kelihos_ver3	2942	2206	736
4	Vundo	475	356	119
5	Simda	42	31	11
6	Tracur	751	563	188
7	Kelihos_ver1	398	298	100
8	Obfuscator.ACY	1228	921	307
9	Gatak	1013	759	254
-	Total	10868	8147	2721

[25] Devido a limitações computacionais, em nossa abordagem, extraímos vetores para unigramas, bigramas e trigramas

[26] No nosso contexto, um termo é um opcode ou operando no formato de byte em hexadecimal

[27] No *one-hot-encoding*, cada classe ou rótulo é representado por vetores com C posições de variáveis binárias, onde para cada classe, apenas uma posição possui o bit 1 "ligado" e as outras posições permanecem com o valor 0. Para C classes obteremos C vetores distintos.

5.2 CONSTRUÇÃO DAS REDES

O algoritmo criado para configuração da topologia da rede autocodificadora (Algoritmo 1) recebe como entrada um vetor $c^{encoder}$, de j posições, onde cada posição j indica uma camada da função codificadora da rede autocodificadora.

Algorithm 1 Criação de uma rede autocodificadora

1: **função** GERARAUTOCODIFICADORA(cEnc, hAct, outAct)
▷ cEnc é o vetor representando as camadas da função codificadora; e hAct é a função de ativação a ser aplicada em todas as camadas ocultas da rede e outAct é a função de ativação a ser aplicada na camada de saída da rede.
2: $mEnc$ e $mDec$ inicialmente não possuem
3: $mEnc \leftarrow gerarCodificadora(mEnc, cEnc, hAct)$
4: $mDec \leftarrow gerarDecodificadora(mDec, cEnc, hAct, outAct)$
5: $mAE \leftarrow rede(mEnc, mDec)$
6: **devolve** mAE
7: **fim função**

O valor preenchido para cada posição no vetor indica a quantidade de neurônios que existirão na camada. Com isso, temos que a primeira posição de $c_0^{encoder}$ representa a camada de entrada da rede que, para facilitar o entendimento, chamaremos de c_{in}. O valor preenchido para c_{in} deve ser definido tendo em mente a dimensionalidade do vetor v_k proveniente do conjunto de dados V_{treino_N} (Figura 5.2).

Algorithm 2 Modelo da função decodificadora de uma rede autocodificadora

1: **função** GERARDECODIFICADORA(mDec, cEnc, hAct, outAct)
▷ mDec é a porção decodificadora da rede que será montada; cEnc é o vetor representando as camadas da função codificadora; hAct é a função de ativação a ser aplicada nas camadas intermediárias da rede e outAct é a função de ativação a ser aplicada na camada de saída da rede.
2: $n \leftarrow tamanho(cEnc)$
3: $cDec \leftarrow copiarItens(cEnc, 0, n-1)$
4: $cDec \leftarrow inverter(cDec)$
5: **para** i ← 0 **até** n-1 **faça**
6: **se** $i < n-1$ **então**
7: $mDec \leftarrow mDec + camadaOculta(cDec[i], hAct)$
8: **senão**
9: $mDec \leftarrow mDec + camadaSaida(cDec[i], outAct)$
10: **fim se**
11: $i \leftarrow i+1$
12: **fim para**
13: **fim função**

Qualquer posição subsequente no vetor, representa uma camada oculta da rede, sendo

a mais profunda H_x. Os valores para cada uma dessas posições no vetor indicam a quantidade de neurônios da camada correspondente na rede (Figura 5.3, Algoritmo 3). O algoritmo monta automaticamente a função de decodificação da rede autocodificadora utilizando o vetor $c^{encoder}$ como base, excluindo a última posição do vetor – H_x – e o utiliza em ordem inversa (Algoritmo 2).

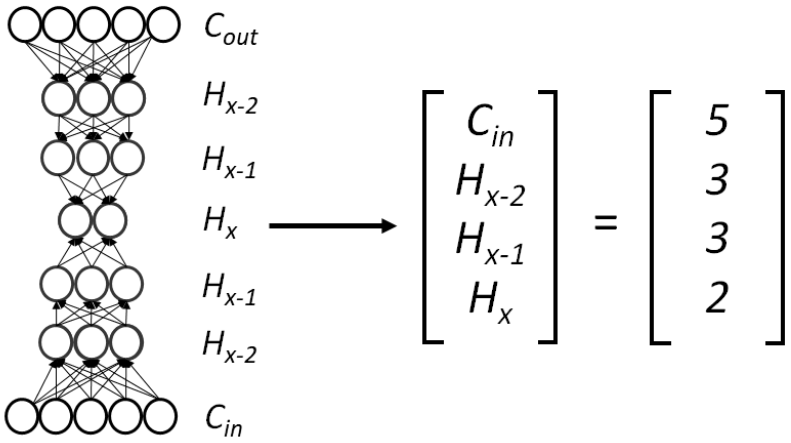

FIG. 5.3: Exemplo de um vetor de configuração para criação de uma rede autocodificadora profunda. Cada posição no vetor representa uma camada da rede, sendo a primeira posição a camada de entrada. Camadas subsequentes representam as camadas internas da rede, fazendo com que o vetor alimentado represente a função de codificação da autocodificadora. Não existe necessidade de informar a função decodificadora já que esta é criada de maneira automática. Os valores para cada posição informam a quantidade de neurônios na camada.

Algorithm 3 Modelo da função codificadora de uma rede autocodificadora

1: **função** GERARCODIFICADORA(mEnc, cEnc, hAct)
▷ mEnc é a porção codificadora da rede que será montada; cEnc é o vetor representando as camadas da função codificadora; e hAct é a função de ativação a ser aplicada em todas as camadas ocultas da rede
2: $n \leftarrow tamanho(cEnc)$
3: $mEnc \leftarrow mEnc + camadaEntrada(cEnc[0])$
4: **para** i ← 1 **até** n-1 **faça**
5: $mEnc \leftarrow mEnc + camadaOculta(cEnc[i], hAct)$
6: $i \leftarrow i + 1$
7: **fim para**
8: **fim função**

Com isso, podemos afirmar que um vetor de j posições produz uma rede autocodificadora de $2j - 1$ camadas, e que o tamanho mínimo de um vetor para produzir a menor rede autocodificadora possível seria $j = 2$.

As funções de ativação utilizadas na rede são definidas por camada, sendo que para camadas ocultas, a função informada sempre será a mesma para cada camada oculta da rede e para a camada de saída, pode-se definir outra função de ativação.

Apenas para exemplificar, temos

$$c^{encoder} = \begin{bmatrix} C_{in} \\ H_{x-2} \\ H_{x-1} \\ H_x \end{bmatrix} = \begin{bmatrix} 5 \\ 3 \\ 3 \\ 2 \end{bmatrix} \tag{5.3}$$

aplicando as regras descritas anteriormente, a função de decodificação seria construída utilizando

$$c^{decoder} = \begin{bmatrix} H_{x-2} \\ H_{x-1} \\ C_{out} \end{bmatrix}, sendo\ C_{out} = C_{in},\ temos\ c^{decoder} = \begin{bmatrix} 3 \\ 3 \\ 5 \end{bmatrix} \tag{5.4}$$

finalizando assim a montagem de uma rede autocodificadora com sete camadas.

5.3 TREINAMENTO E VALIDAÇÃO

Neste trabalho, para resolver o problema de classificação de *malware*, aplicamos dois tipos de redes neurais profundas com objetivos complementares. Inicialmente, treinamos com o conjunto de dados construído uma rede autocodificadora com o objetivo de aprendizado de características e, em alguns casos, redução de dimensionalidade (GOODFELLOW et al., 2016). O treinamento dessas redes é não supervisionado, portanto a rede é treinada em dados não rotulados. Acreditamos que padrões retirados diretamente do código malicioso sejam aprendidos pela rede, que terá codificada em sua camada mais interna, uma nova representação dos dados apresentados, uma espécie de assinatura do *malware*. Anteriormente, redes de aprendizado profundo só conseguiam efetivamente serem treinadas com o emprego de um algoritmo guloso de pré-treinamento não supervisionado camada-a-camada e posterior ajuste dos pesos com treinamento supervisionado[28] (ERHAN et al., 2010; GOODFELLOW et al., 2016). O algoritmo é guloso pois procura otimizar cada camada da rede profunda independentemente. Finalmente, realiza um ajuste na rede por completo utilizando desta vez o treinamento supervisionado. O algoritmo atua como um regularizador e também como uma forma de inicialização de parâmetros da rede.

Todavia, a criação de novas técnicas de regularização e inicialização dos pesos de re-

[28]do inglês, *Greedy layer-wise unsupervised pre-training with supervised fine-tuning*

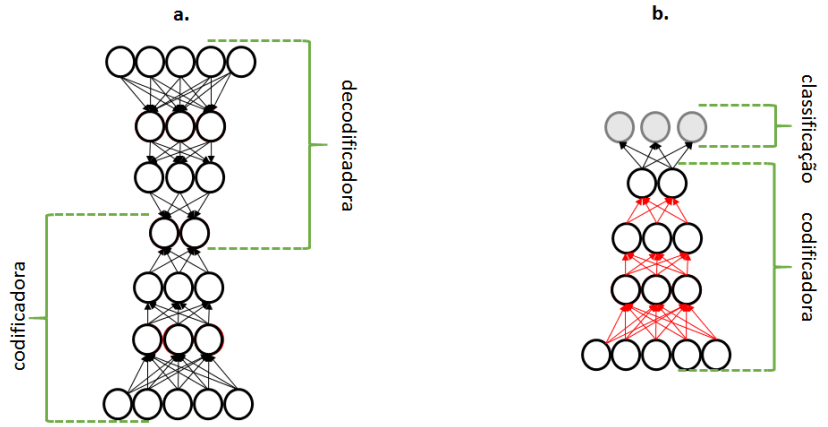

FIG. 5.4: Esquemática do pré-treinamento não supervisionado na visão da topologia das redes neurais. Em **a.** temos uma autocodificadora profunda, com destaque para suas funções codificadora e decodificadora. Já em **b.**, vemos uma rede MLP construída com base na função codificadora da rede autocodificadora e a camada adicionada para treinamento da fase supervisionada – classificação. A imagem coloca em destaque (vermelho) os pesos que foram pré-inicializados pela autocodificadora.

des neurais tornaram o uso deste algoritmo menos comum. Porém, ainda podem existir vantagens de se utilizar o pré-treinamento não supervisionado. Em nossa abordagem, decidimos realizar o pré-treinamento não-supervisionado da rede profunda por completo, não realizando o pré-treinamento camada-a-camada. A tarefa que a rede autocodificadora profunda treinada de maneira não supervisionada tenta executar é a captura da forma da distribuição das amostras observadas, o que acreditamos que possa facilitar a tarefa de classificação dos *malware*, isto é, a representação aprendida na tarefa não supervisionada possa ser útil para a tarefa supervisionada utilizando o mesmo domínio de entrada (GOODFELLOW et al., 2016).

Como o objetivo final é a classificação de amostras de *malware* dentre um número de famílias pré-estabelecido, a próxima etapa de nossa abordagem é a utilização da função codificadora da rede autocodificadora como base de uma rede de múltiplas camadas para a classificação. Essa etapa foi favorecida pela semelhança entre as redes escolhidas, sendo necessário para a criação da nova rede apenas a inclusão de uma camada de classificação no topo da função codificadora que fora previamente treinada (Figura 5.4). Por fim, essa nova rede é treinada e validada com o conjunto de amostras rotuladas, encerrando assim o processo.

Para o treinamento das redes, configura-se a taxa de aprendizado do Gradiente Descendente Estocástico, o número de épocas de treinamento, o tamanho das *minibatches* e a função de custo ou erro a ser utilizada. Também se configura qualquer outra condição

Algorithm 4 Treinamento de uma rede autocodificadora

função TREINARAUTOCODIFICADORA(mAE,V_{treino_N}, V_{teste_N}, f_{custo}, $txAprendizado$, $minibatch$, $epocas$)
  $V_{treino_{ajust}} \leftarrow removeRotulos(V_{treino_N})$
  $V_{teste_{ajust}} \leftarrow removeRotulos(V_{teste_N})$
  enquanto nenhuma condição de parada for alcançada **faça**
    para i $\leftarrow 1$ **até** epocas **faça**
      repita
        $batch \leftarrow extraiSubconjunto(V_{treino_{ajust}}, minibatch)$
        $treinar(mAE, batch, txAprendizado, f_{custo})$
      até que $batch = \emptyset$
      $validar(mAE, V_{teste_{ajust}}, txAprendizado, f_{custo})$
      $i \leftarrow i + 1$
    fim para
  fim enquanto
fim função

de parada antecipada – *early stopping* – que se faça necessária. A validação do modelo treinado ocorre ao final de cada época. O treinamento cessa quando alguma condição de parada é alcançada (Algoritmo 4). O treinamento do classificador ocorre posteriormente, se beneficiando do que foi aprendido pela autocodificadora. O treinamento é semelhante, porém, agora, utiliza o conjunto de treinamento rotulado (Algoritmo 5).

Algorithm 5 Treinamento da rede utilizando a função codificadora já treinada adicionada de uma camada de classificação

função TREINARCLASSIFICADOR(mAE,V_{treino_N}, V_{teste_N}, f_{custo}, $txAprendizado$, $minibatch$, $epocas$)
  $fCodif \leftarrow extraiCodificadora(mAE)$
  $classif \leftarrow preInicializa(fCodif, classif)$
  enquanto nenhuma condição de parada for alcançada **faça**
    para i $\leftarrow 1$ **até** epocas **faça**
      repita
        $batch \leftarrow extraiSubconjunto(V_{treino_N}, minibatch)$
        $treinar(classif, batch, txAprendizado, f_{custo})$
      até que $batch = \emptyset$
      $validar(classif, V_{teste_{ajust}}, txAprendizado, f_{custo})$
      $i \leftarrow i + 1$
    fim para
  fim enquanto
fim função

5.4 MÉTRICAS

De acordo com Sokolova e Lapalme (2009), a exatidão de uma classificação pode ser avaliada ao computar-se o número de amostras de classe corretamente reconhecidos (verdadeiros positivos), o número de exemplos corretamente reconhecidos que não pertencem a classe indicada (verdadeiros negativos) e exemplos que ou foram incorretamente atribuídos a uma classe (falsos positivos) ou que não foram reconhecidos como exemplos de uma classe (falsos negativos). Diversas medidas de performance podem ser calculadas com base nesses valores. Exemplos das mais utilizadas são a **acurácia**, **precisão**, **revocação**(ou **abrangência**) e a ***Medida* $\mathbf{F_1}$**. Nesta seção, detalharemos alguns dos métodos utilizados para avaliação dos modelos neste trabalho.

5.4.1 MATRIZ DE CONFUSÃO

Uma das maneiras de expor a relação entre como as amostras foram previstas e como as amostras deveriam ser realmente classificadas por um classificador é utilizando uma matriz de confusão. Tal matriz consiste de uma estrutura onde são exibidas as contagens de verdadeiros positivos (VP), verdadeiros negativos (VN), falsos positivos (FP) e falsos negativos (FN) obtidos por um classificador. Pode ser utilizada para exibição tanto dos resultados de classificações binárias quanto de multi-classe (Figura 5.5) (SOKOLOVA; LAPALME, 2009)

5.4.2 PRECISÃO

A precisão é uma medida da qualidade das previsões de um classificador. É calculada utilizando o número de amostras classificadas corretamente (verdadeiros positivos) dividido pelo número de amostras que foram classificadas como positivas pelo classificador (número de verdadeiros positivos e os falsos positivos) (SOKOLOVA; LAPALME, 2009; MANNING et al., 2008a). É calculada pela fórmula

$$Prec = \frac{VP}{VP + FP} \tag{5.5}$$

onde VP é a quantidade de verdadeiros positivos e FP a quantidade de falsos positivos.

5.4.3 ABRANGÊNCIA OU REVOCAÇÃO

A revocação ou Abrangência mede a eficácia de um classificador em identificar corretamente amostras relevantes. É calculada utilizando o número de amostras que foram

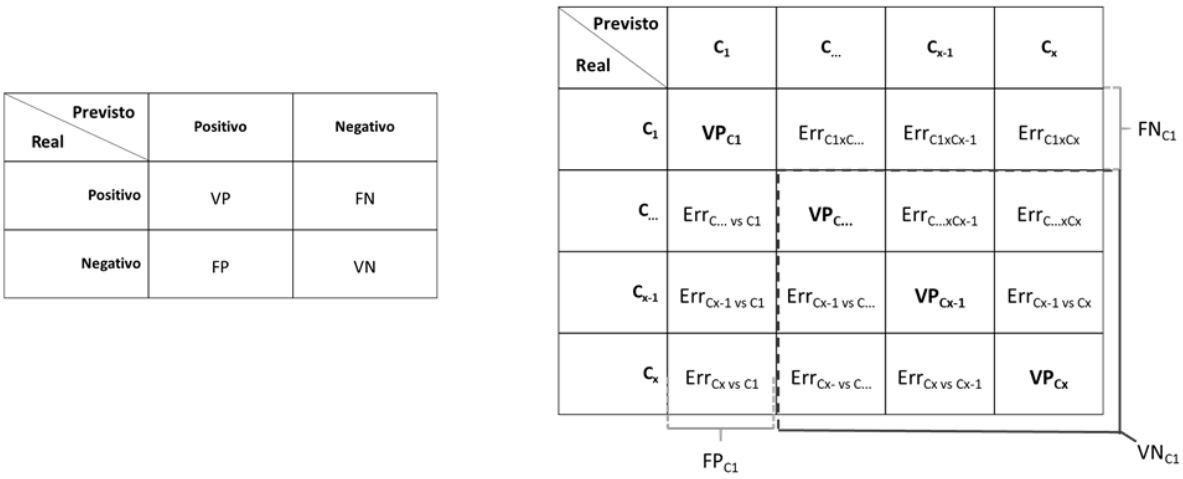

FIG. 5.5: À esquerda, a montagem de uma matriz de confusão para classificação binária e à direita para a exibição de resultados multi-classe. No caso multi-classe, a diagonal principal apresenta os resultados dos verdadeiros positivos para todas as classes. Os erros em uma determinada linha da matriz representam os falsos negativos da classe associada a linha. Os erros em uma determinada coluna representam os falsos positivos da classe associada a coluna. Verdadeiros Negativos de uma classe são todos os outros resultados que não estejam nem na linha nem na coluna associada a classe.

classificadas corretamente (verdadeiros positivos) dividido pelo número total de amostras de uma determinada classe no conjunto de dados (verdadeiros positivos e falsos negativos) (SOKOLOVA; LAPALME, 2009; MANNING et al., 2008a). É calculada pela fórmula

$$Rev = \frac{VP}{VP + FN} \tag{5.6}$$

onde VP é a quantidade de verdadeiros positivos e FN é quantidade de falsos negativos.

5.4.4 MEDIDA Fβ

Duas taxas que são interessantes de serem medidas para o acompanhamento da performance de um modelo são a Precisão e a Revocação ou Abrangência.

A Medida F é uma medida que realiza o balanceamento essas duas medidas, a Precisão e a Revocação, sendo a sua média harmônica ponderada, definida de maneira geral por Manning et al. (2008a) como

$$F_\beta = \frac{(\beta^2 + 1)PR}{\beta^2 P + R} \tag{5.7}$$

onde P é a medida da precisão, R é a medida da revocação, $\beta^2 = \frac{1-\alpha}{\alpha}$ e $\beta^2 \in [0, \infty]$. A configuração mais utilizada para a medida pondera de maneira igual tanto a precisão quanto a revocação, ou seja, onde $\beta = 1$, é chamada de medida F_1, cuja fórmula pode ser

reescrita de forma simplificada como:

$$F_1 = 2\frac{PR}{P+R} \tag{5.8}$$

No caso de problemas multi-classe, existem duas formas de generalizar a medida F_1: A **Média-Macro $\mathbf{F_1}$** e a **Média-Micro $\mathbf{F_1}$** (MURPHY, 2012; SOKOLOVA; LAPALME, 2009). A **Média-Macro $\mathbf{F_1}$** é definida como a média aritmética simples de todas as medições de F_1 para as possíveis classes do problema. A fórmula pode ser escrita como:

$$\mathrm{Macro}_{\mathrm{F}_1} = \frac{\sum_{c=1}^{C} \mathrm{F}_1(c)}{C} \tag{5.9}$$

onde C representa o número máximo de classes ou rótulos do problema e $F_1(c)$ representa a medida F_1 para uma determinada classe c. Já a **Média-Micro $\mathbf{F_1}$** é calculada utilizando-se o somatório de falsos negativos, falsos positivos, verdadeiros negativos e verdadeiros positivos de todas as classes do problema, prosseguindo então para o cálculo da medida. A diferença entre as duas medidas é que utilizando a $Macro_{F_1}$, consideramos que todas as classes possuem a mesma relevância, ou seja, possuem o mesmo peso. Ao utilizar a $Micro_{F_1}$, favorece-se a classe com o maior valor (MURPHY, 2012; SOKOLOVA; LAPALME, 2009).

Usualmente, programas de detecção ou classificação de *malware* devem possuir um número baixo de falsos positivos e um número ainda mais baixo de falsos negativos. Como a medida F1 é uma média entre duas outras taxas que são influenciadas por falsos positivos e negativos – precisão e revocação – e a idéia de possuir apenas uma medida que consiga expressar a qualidade do classificador, em um problema multi-classe, independentemente do desbalanceamento das classes no conjunto de dados, optou-se então por utilizar a média $Macro_{F_1}$.

6 RESULTADOS EXPERIMENTAIS

Neste capítulo são detalhadas todas as configurações da fase experimental deste trabalho e os resultados obtidos na aplicação da abordagem definida no capítulo 5.

6.1 ANÁLISE DE SENSIBILIDADE

Neste trabalho serão realizados experimentos visando a avaliação da utilização da abordagem descrita no capítulo 5, abordagem essa que se fundamenta na não utilização de conhecimentos prévios que auxiliem no problema da classificação de *malware*. Com isso, espera-se validar a abordagem e verificar a performance dos modelos gerados e o impacto que mudanças em suas configurações podem ocasionar, como por exemplo, ganho ou perda de desempenho. No intuito de gerarmos um número suficiente de modelos para esta investigação, definimos uma regra de formação para as diversas topologias de redes a serem experimentadas. Para cada conjunto de dados V_{treino_N} e V_{teste_N}, onde N significa o N-grama utilizado na geração do conjunto de dados e (v_k, y_k) uma tupla arbitrária contida em um desses conjuntos, as redes utilizarão as seguintes regras de formação: A camada de entrada da rede terá o número de neurônios definido pelo tamanho do vetor de características do conjunto de dados, ou seja, o tamanho de v_k. Redes com apenas uma camada oculta serão criadas obedecendo a seguinte regra em relação ao número de neurônios por camada: Multiplica-se um fator x pelo número de neurônios na camada de entrada, obtendo-se assim o número de neurônios na camada oculta. Os fatores foram definidos como $X = \{0.1, 0.2, 0.3, \ldots, 1.9, 2.0\}$. A escolha desses fatores teve como objetivo produzir um número variado de redes autocodificadoras tanto subcompletas como sobrecompletas. A profundidade máxima a ser investigada para as redes classificadoras será de 10 camadas ocultas. Como previamente mencionado, para se produzir uma rede classificadora de j camadas, necessitamos construir uma rede autocodificadora de $2j - 1$ camadas. Para decidir o número de neurônios para cada camada subsequente até a décima, realizamos o cálculo da razão entre o número de neurônios da primeira camada oculta da rede e o número de neurônios da camada de saída da rede classificadora utilizando a fórmula da progressão aritmética de forma a determinarmos a quantidade de neurônios para cada camada até a última camada escondida. A camada de saída da rede classificadora tem o número de neurônios definido pela quantidade de rótulos possíveis no

conjunto de dados.

6.2 CONJUNTO DE DADOS

Conforme apresentado na seção 5.1.3, foram gerados conjuntos de dados de treinamento e teste para cada n-grama a ser experimentado. Os vetores de características foram gerados pelo *TfIdfVectorizer* da biblioteca Scikit-learn, configurado para gerar vetores de unigramas, bigramas e trigramas. Os vetores já são gerados com a ponderação pelo TF-IDF.

A partir de todas as amostras do conjunto de treinamento do conjunto de dados de referência, foram extraídos 96 OpCodes para unigramas e 9216 para bigramas. Para trigramas, devido ao grande número de combinações de OpCodes (884736 possíveis), utilizamos uma regra de seleção dos 10000 "melhores" termos, considerando o valor calculado pelo TF-IDF para cada termo, do maior para o menor valor, isto é, ordenando-se do termo com maior peso para o termo com o menor peso.

6.2.1 TREINAMENTO E AVALIAÇÃO DOS MODELOS

De forma a controlar o número de redes a serem experimentadas, dos **vinte** modelos de uma camada oculta inicialmente configurados, apenas os **cinco** melhores serão escolhidos para continuarem a ser testados com o incremento de novas camadas ocultas. Para isso, utilizou-se como critério a medida $\text{Macro}_{\text{F}_1}$ (seção 5.4.4).

Sendo assim, foram criados 100 experimentos, divididos da seguinte forma entre os conjuntos de dados: Para unigramas, 65 experimentos, sendo 65 redes autocodificadoras e 65 redes multicamadas e para bigramas, 35 experimentos, sendo 35 redes autocodificadoras e 35 redes multicamadas.

As configurações pertinentes às redes neurais foram padronizadas para todos os experimentos. Escolheu-se apenas uma função de ativação para todas as camadas ocultas da rede e uma função de ativação para a camada de saída da rede. As configurações relativas ao treinamento e validação das redes também foram padronizadas, como o tamanho do *batch* de amostras, o otimizador e seus parâmetros, função de erro ou custo e a quantidade de épocas de treinamento. Detalhes dessas configurações podem ser observados na tabela 6.1. As configurações de parada antecipada estão detalhadas na tabela 6.2. O mesmo critério de parada antecipada foi utilizado para todas as autocodificadoras: Aguardar até 100 épocas sem qualquer melhora na medição da função de custo – diminuição do erro em pelo menos 0.01 – do modelo no momento de sua validação. Já para as redes

de múltiplas camadas, configurou-se: Aguardar até 100 épocas sem qualquer melhora na medição da acurácia, ou seja, um aumento de 0.01 na acurácia do modelo no momento do treinamento.

TAB. 6.1: Tabela de configurações de parâmetros dos experimentos

Configuração	**Autocodificadora**	**MLPs**
Ativação (Camada Oculta)	ReLU	ReLU
Ativação (Camada de Saída)	ReLU	Sigmoid
Função de Custo/Erro	Erro Quadrático Médio	Entropia Cruzada
Otimizador	SGD (Taxa de Aprendizado: 0.01)	SGD (Taxa de Aprendizado: 0.01)
Tamanho do Batch	32	32
Épocas	1000	1000

TAB. 6.2: Tabela de configurações da Parada Antecipada nos experimentos

Arquitetura	**Medida**	Δ_{min}	**Paciência**	**Fase**
Autocodificadora	função de erro	0.01	100	Validação
MLP	acurácia	0.01	100	Treinamento

Como função de ativação foi escolhida a Unidade Linear Retificada (ReLU), por ser esta uma função de ativação recomendada para o uso em redes neurais mais modernas (GOODFELLOW et al., 2016; GLOROT et al., 2011). Para a classificação foi escolhida a função Sigmoide Logística. As funções de custo escolhidas foram as que melhor se comportaram para o tipo do problema enfrentado, baseado em experimentos anteriores. Para as redes autocodificadoras utiliza-se o Erro Quadrático Médio e para as redes classificadoras a Entropia Cruzada.

6.3 SOFTWARE

Os experimentos foram programados em Python 3.5, utilizando a biblioteca Keras (CHOLLET et al., 2015) versão 2.x, utilizando como backend o Tensorflow (ABADI et al., 2016) na versão 1.3. Para calcular o desempenho dos modelos, matriz de confusão e outras medidas de performance, foi utilizada a biblioteca Pandas ML[29] na versão 0.4.

6.4 ANÁLISE DOS RESULTADOS

No intuito de melhor avaliar os resultados dos modelos treinados, apresentaremos os resultados utilizando os conjuntos de dados constituídos de vetores de Unigramas e Bigramas separadamente e posteriormente, realizaremos uma avaliação mais aprofundada em redes

[29]https://github.com/pandas-ml/pandas-ml

que obtiveram o melhor e pior desempenho geral. Utilizamos a média $Macro_{F_1}$ como medida principal da qualidade do modelo, porém, exibimos também os resultados do F_1 da classe mais frequente no conjunto de dados – Kelihos_ver3 – bem como os resultados do F_1 da classe mais rara – Simda. Os valores exibidos foram medidos durante a fase de validação do modelo, ou seja, durante a execução do modelo já treinado utilizando 25% do conjunto de dados não utilizados no treinamento.

6.4.1 RESULTADO DOS EXPERIMENTOS NO CONJUNTO DE UNIGRAMAS

Foram treinadas no total 65 redes autocodificadoras que pré-inicializaram 65 redes "multicamadas". Os primeiros modelos treinados e validados foram os de redes de apenas **1 camada oculta**, totalizando 20 redes – 10 advindas de redes autocodificadoras subcompletas e 10 de sobrecompletas.

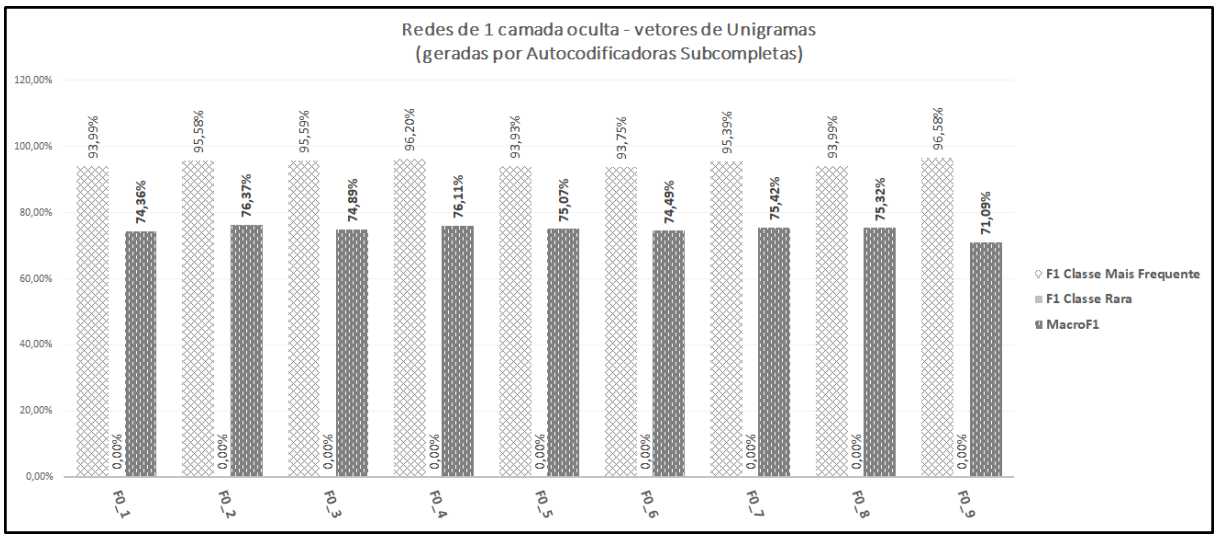

FIG. 6.1: Resultados de redes de 1 camada oculta para o conjunto de dados de unigramas. Apenas modelos advindos de autocodificadoras subcompletas.

Podemos observar no gráfico (Figura 6.1), os resultados obtidos pelas redes pré-inicializadas por autocodificadoras subcompletas. Verifica-se que esses modelos obtiveram $Macro_{F_1}$ em uma faixa de $71,09\%$ (Rede F0_9 com o menor valor) a $76,37\%$ (Rede F0_2 com o maior valor). Já os modelos de redes advindas de autocodificadoras sobrecompletas (gráfico na Figura 6.2) obtiveram medidas de $Macro_{F_1}$ em uma faixa de $74,33\%$ (Rede F1_8 com o menor valor) a $76,83\%$ (Rede F1_5 com o maior valor). No geral, todos os modelos obtiveram valores altos da medida F_1 para a classe mais frequente, com resultados variando de $92,87\%$ (Rede F1_8) a $97,02\%$ (Rede F1_5). Observa-se que nenhum modelo utilizando apenas uma camada oculta aprendeu a reconhecer os padrões da classe

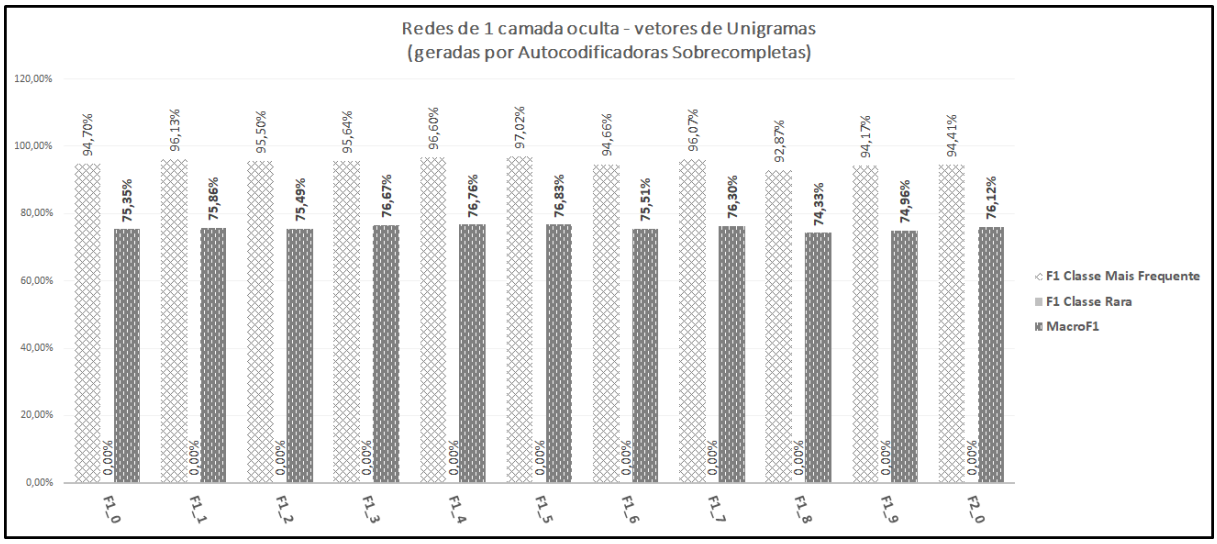

FIG. 6.2: Resultados de redes de 1 camada oculta para o conjunto de dados de unigramas. Apenas modelos advindos de autocodificadoras subcompletas.

mais rara de *malware*, não acertando nenhuma das amostras apresentadas, obtendo assim **zero** para a medida F_1 nesta categoria.

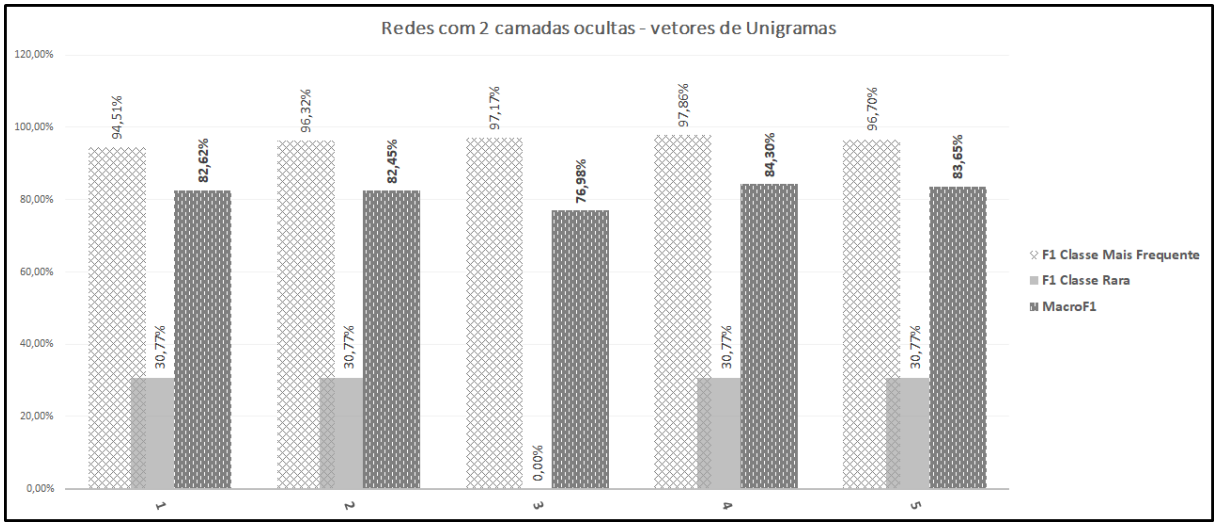

FIG. 6.3: Resultados de redes de 2 camadas ocultas para o conjunto de dados de unigramas.

Redes com **2 camadas ocultas** (gráfico na Figura 6.3) obtiveram medições de $Macro_{F_1}$ superiores às redes anteriores, variando de $76,98\%$ para a menor medição (Rede 3) a $84,30\%$ para a maior medição (Rede 4). O mesmo comportamento pode ser observado em relação a medida da classe mais frequente, que continua se mantendo alta, variando de $94,51\%$ (Rede 1) e $97,86\%$ (Rede 4). Modelos com duas camadas ocultas já conseguiram diferenciar algumas amostras da classe mais rara do conjunto de dados, e mesmo com capacidades diferentes, 4 redes obtiveram o mesmo resultado da medida F_1,

30, 77%, exceto para Rede 3, que obteve **zero**.

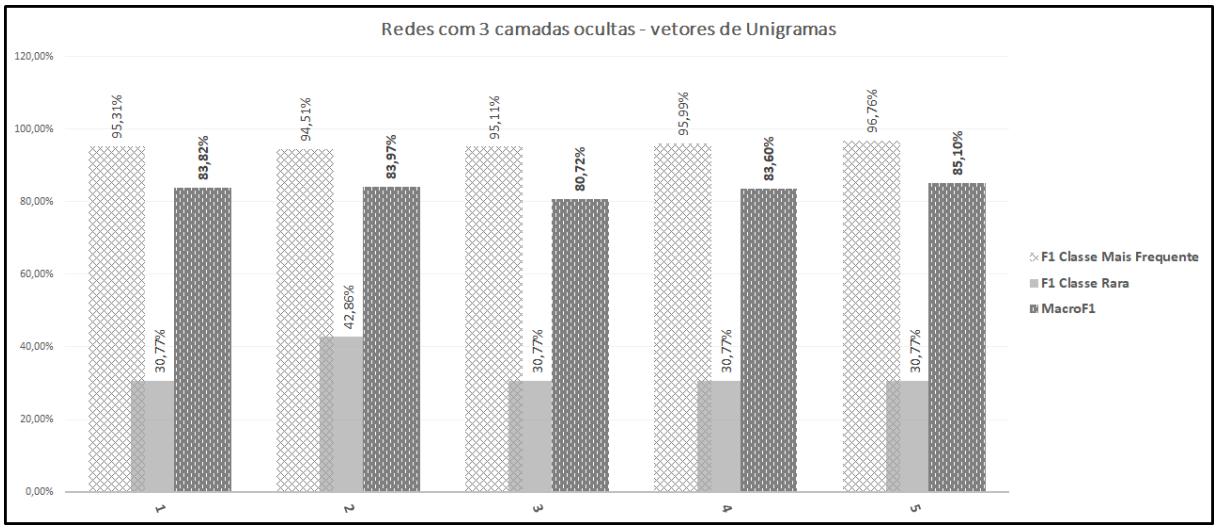

FIG. 6.4: Resultados de redes de 3 camadas ocultas para o conjunto de dados de unigramas.

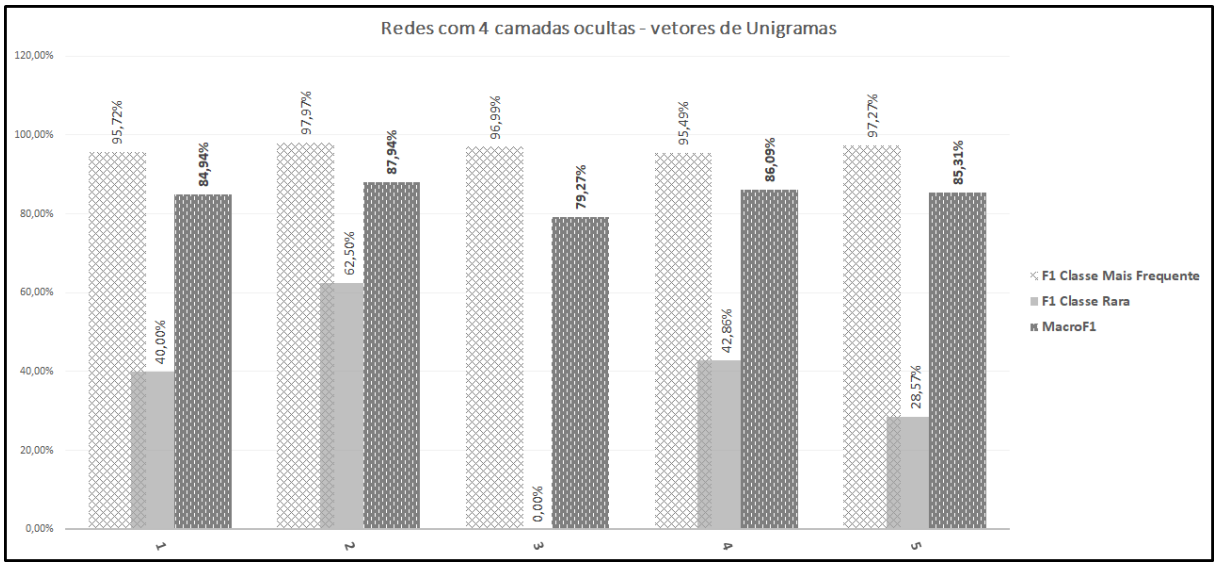

FIG. 6.5: Resultados de redes de 4 camadas ocultas para o conjunto de dados de unigramas.

As redes com **3 camadas ocultas** conseguiram melhor performance em relação as avaliadas anteriormente, com todas as redes ficando acima de 80% para $Macro_{F_1}$ (gráfico na Figura 6.4), com a pior rede medindo 80, 72% (Rede 3) e a melhor 85, 10% (Rede 5). Todas as redes obtiveram algum sucesso com amostras da classe mais rara. O destaque nesta série de experimentos vai para a Rede 2, que obteve o maior valor de F_1, obtendo 42, 86%. Novamente os valores medidos para a classe mais frequente ficaram acima de 90%, sendo a Rede 2 a que mediu o menor valor, 94, 51%, e a Rede 5 que mediu o maior

valor $96,76\%$.

Já nos resultados de redes de **4 camadas ocultas** (gráfico na Figura 6.5), podemos destacar a Rede 2, que obteve os maiores valores para todas as categorias medidas, $Macro_{F_1} = 87,94\%$, para a classe mais frequente $F_1 = 97,97\%$ e para a classe mais rara $F_1 = 62,50\%$. A Rede 1 obteve a pior medição para $Macro_{F_1}$, $79,27\%$. A pior rede considerando a classe mais frequente foi a Rede 4, com $F_1 = 95,49\%$. Para a classe mais rara, o pior resultado medido foi o da Rede 3 que obteve $F_1 = 0$.

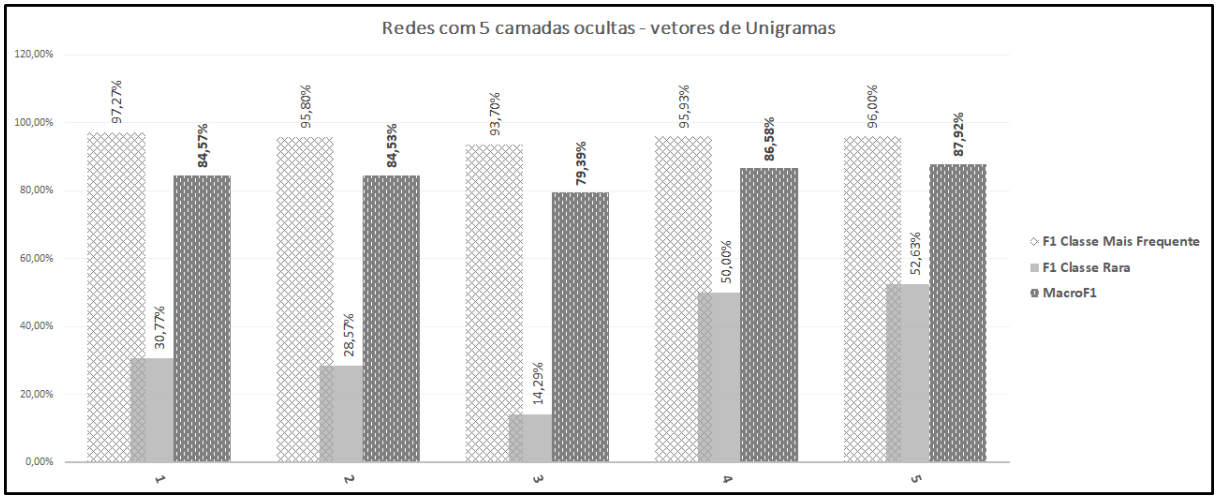

FIG. 6.6: Resultados de redes de 5 camadas ocultas para o conjunto de dados de unigramas.

Com redes de **5 camadas ocultas** (gráfico na Figura 6.6), verificamos o $Macro_{F_1}$ variando de $79,39\%$ (pior rede, Rede 3) a $87,92\%$ (melhor rede, Rede 5). O melhor resultado considerando a classe mais rara foi o da Rede 5, e o pior pela Rede 3, respectivamente $52,63\%$ e $14,29\%$. Para a classe mais frequente, obtivemos valores entre $93,70\%$ e $97,27\%$, respectivamente a Rede 3 e Rede 1.

Dentre os modelos de **6 camadas ocultas** (gráfico na Figura 6.7), destaca-se a Rede 5, com $Macro_{F_1} = 86,89\%$, para a classe mais frequente $F_1 = 98,10\%$ e para a classe mais rara, $F_1 = 37,50\%$. O menor valor F_1 medido para a classe mais frequente do conjunto de dados foi $95,03\%$ na Rede 4, e para a classe mais rara, o $F_1 = 16,67\%$ na Rede 3. No caso da classe mais frequente no conjunto de dados, medimos valores de F_1 para classe mais frequente variando de $95,03\%$ (menor valor para Rede 4) a $98,10\%$ (maior valor para Rede 5) e para a classe mais rara, F_1 variando de $16,67\%$ (Rede 3) até $37,50\%$ (Rede5).

Dos modelos com **7 camadas ocultas** (gráfico na Figura 6.8), obtivemos a medida $Macro_{F_1}$ variando de $71,51\%$ (menor valor medido para Rede 2) e $85,51\%$ (maior valor medido para Rede 5). Para a classe mais rara, o F_1 variou entre $15,38\%$ (menor valor

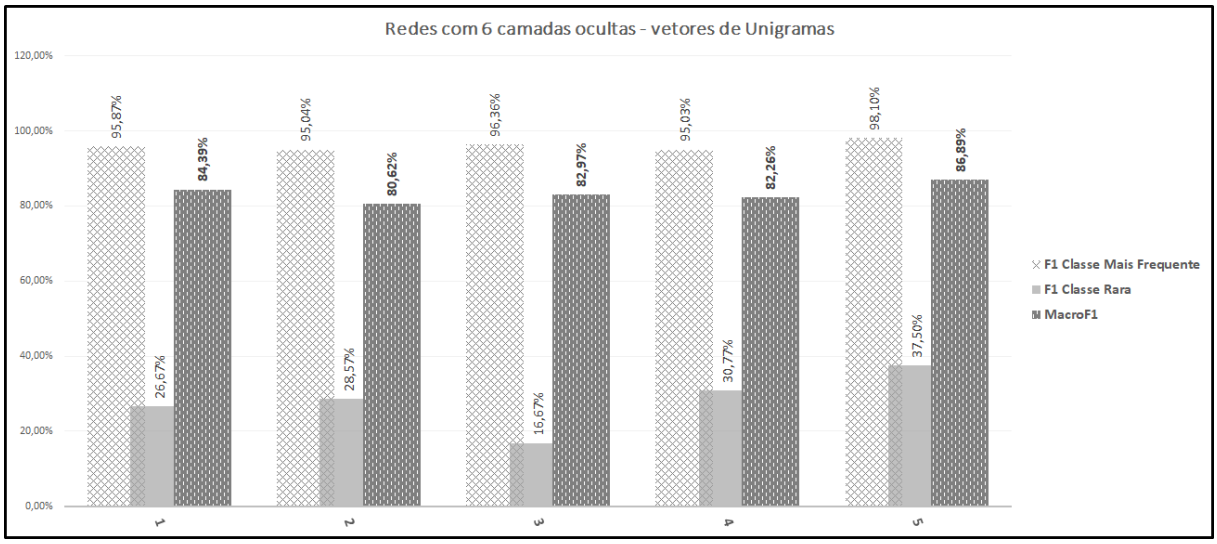

FIG. 6.7: Resultados de redes de 6 camadas ocultas para o conjunto de dados de unigramas.

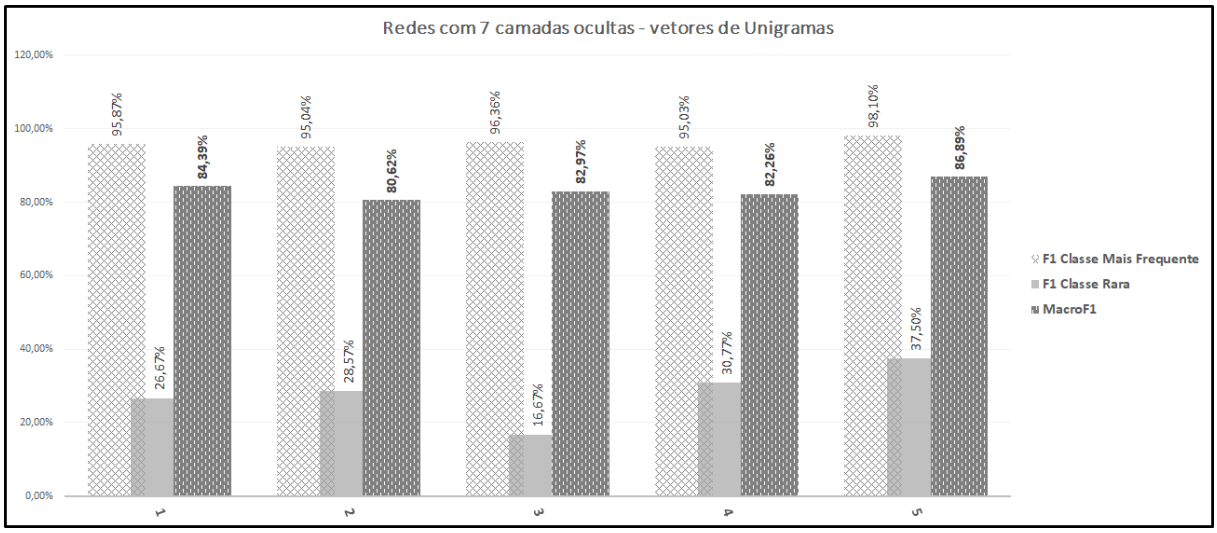

FIG. 6.8: Resultados de redes de 7 camadas ocultas para o conjunto de dados de unigramas.

medido para a Rede 2) e $30,77\%$ (maior valor medido para a Rede 4 e 5). No caso da classe mais frequente, F_1 variou entre $93,14\%$ (menor valor medido para a Rede 4) e $97,22\%$ (maior valor medido para a Rede 1).

Podemos destacar dentre os modelos de **8 camadas ocultas** (gráfico na Figura 6.9) a Rede 5 , que obteve o melhor valor de $Macro_{F_1} = 89,69\%$ e o melhor valor de F_1 para a classe mais rara, $66,67\%$. Destaca-se também a Rede 3, que obteve o pior resultado nas três medidas, com $Macro_{F_1} = 79,39\%$, para a classe mais rara um $F_1 = 0$ e para a classe mais frequente $F_1 = 94,81\%$.

Dentre os experimentos executados com redes de **9 camadas ocultas**, a Rede 4

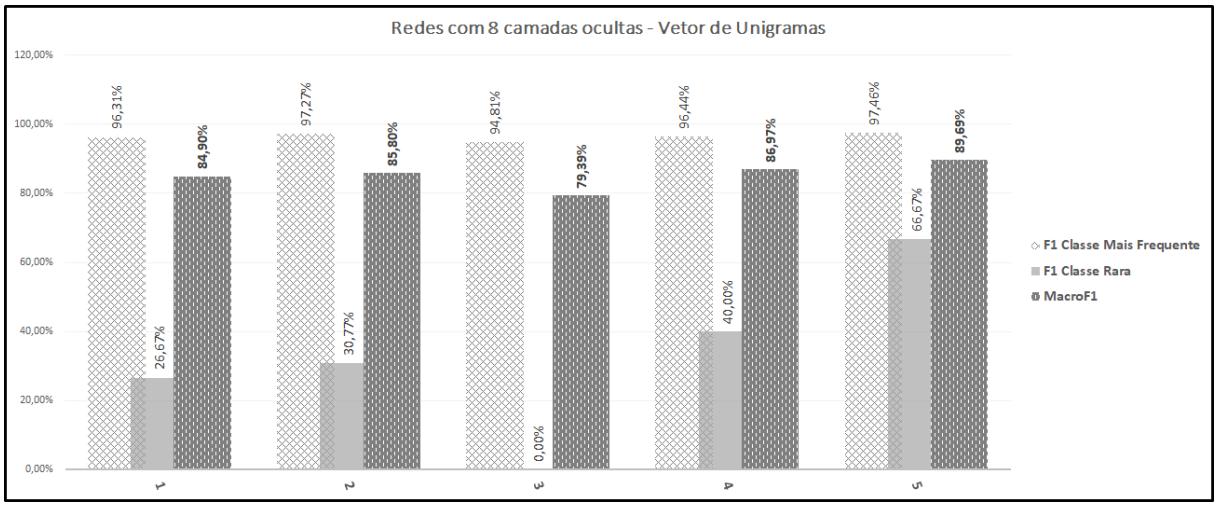

FIG. 6.9: Resultados de redes de 8 camadas ocultas para o conjunto de dados de unigramas.

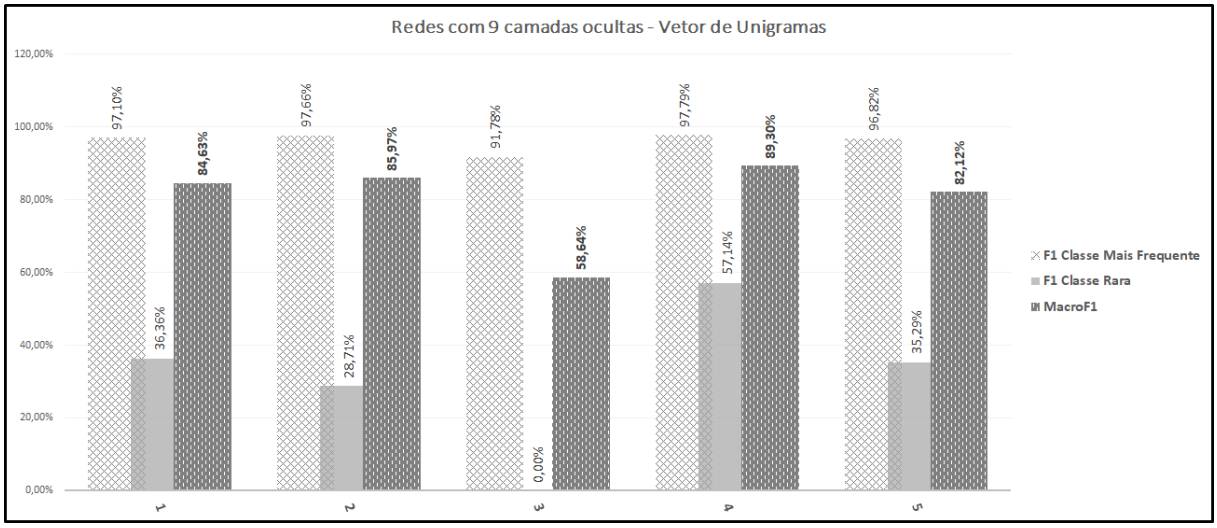

FIG. 6.10: Resultados de redes de 9 camadas ocultas para o conjunto de dados de unigramas.

obteve os melhores valores para as três métricas, $Macro_{F_1} = 89,30\%$, $F_1 = 97,79\%$ para classe mais frequente e $F_1 = 57,14\%$ para a classe mais rara. A Rede 3 obteve os piores resultados, com $Macro_{F_1} = 58,64\%$, $F_1 = 91,78\%$ para classe mais frequente e $F_1 = 0$ para a classe mais rara. As outras redes obtiveram resultados acima de 80% para $Macro_{F_1}$, acima de 96% para a classe mais frequente e variou de $28,71\%$ (Rede 2) a $36,36\%$ (Rede 1) para o F_1 com a classe mais rara.

É interessante notar que apesar da grande quantidade de camadas e de neurônios no conjunto experimental de redes com **10 camadas ocultas** (gráfico na Figura 6.11), observamos três redes (Rede 3, 4 e 5) que obtiveram os piores valores nas métricas, com $Macro_{F_1} = 2,76\%$ e **zero** medido para F_1, tanto para a classe mais frequente como para

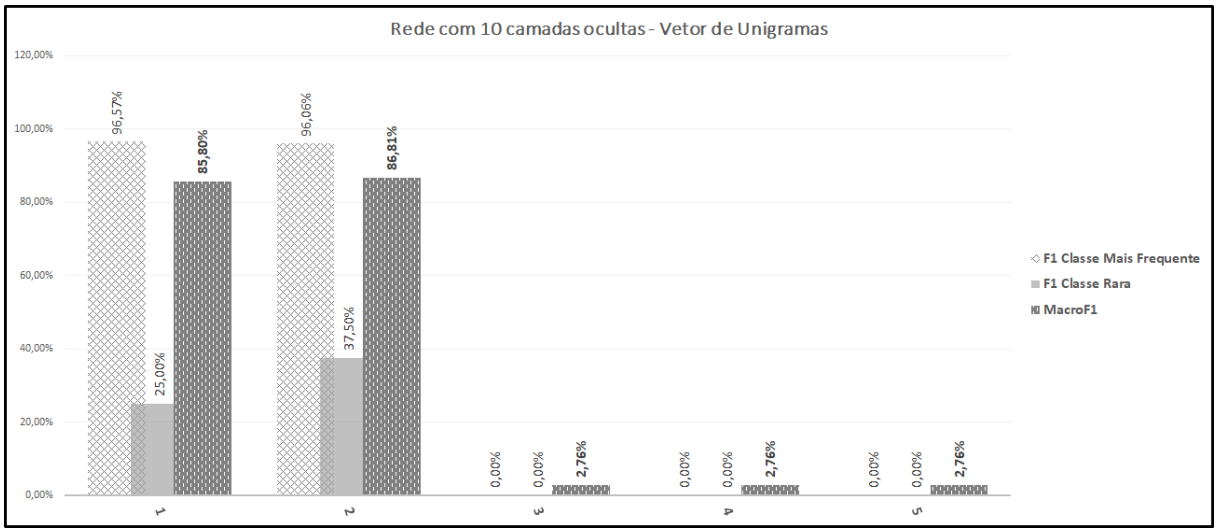

FIG. 6.11: Resultados de redes de 10 camadas ocultas para o conjunto de dados de unigramas.

a classe mais rara. A Rede 2 obteve o maior valor de $Macro_{F_1}$ e de F_1 para a classe mais rara, respectivamente $86,81\%$ e $37,50\%$. O maior valor medido para a classe mais frequente, $F_1 = 96,57\%$, foi obtido pela Rede 1.

6.4.2 RESULTADO DOS EXPERIMENTOS NO CONJUNTO DE BIGRAMAS

Foram treinadas no total 35 redes autocodificadoras que pré-inicializaram 35 redes "multicamadas". Os primeiros modelos treinados e validados foram os de redes de apenas **1 camada oculta**, totalizando 20 redes – 10 advindas de redes autocodificadoras subcompletas e 10 de sobrecompletas. Observa-se nos gráficos (Figura 6.12 e 6.13), que nenhuma rede construída com apenas **1 camada oculta** conseguiu distinguir os padrões da classe mais rara do conjunto de dados.

Das redes pré-inicializadas através de redes autocodificadoras subcompletas, verifica-se que esses modelos obtiveram $Macro_{F_1}$ variando de $82,42\%$ (Rede F0_1) a $82,61\%$ (Rede F0_3), com o F_1 da classe mais frequente variando de $98,92\%$ (Rede F0_2) a $99,32\%$ (Rede F0_6). Das pré-inicializadas através de redes sobrecompletas, observa-se $Macro_{F_1}$ variando de $82,33\%$ (Rede F1_0) a $82,64\%$ (Rede F1_4) e para a classe mais frequente, F_1 medindo de $98,92\%$ (Rede F1_0) a $99,32\%$ (Rede F2_0).

Dentre os modelos construídos com **2 camadas ocultas** (gráfico na Figura 6.14), destaca-se a Rede 5, que obteve os maiores valores nas três medições, $Macro_{F_1} = 92,93\%$, para a classe mais rara, $F_1 = 66,67\%$ e para a mais frequente, $F_1 = 99,59\%$. Todos os modelos treinados conseguiram identificar amostras da classe mais rara, medindo o F_1

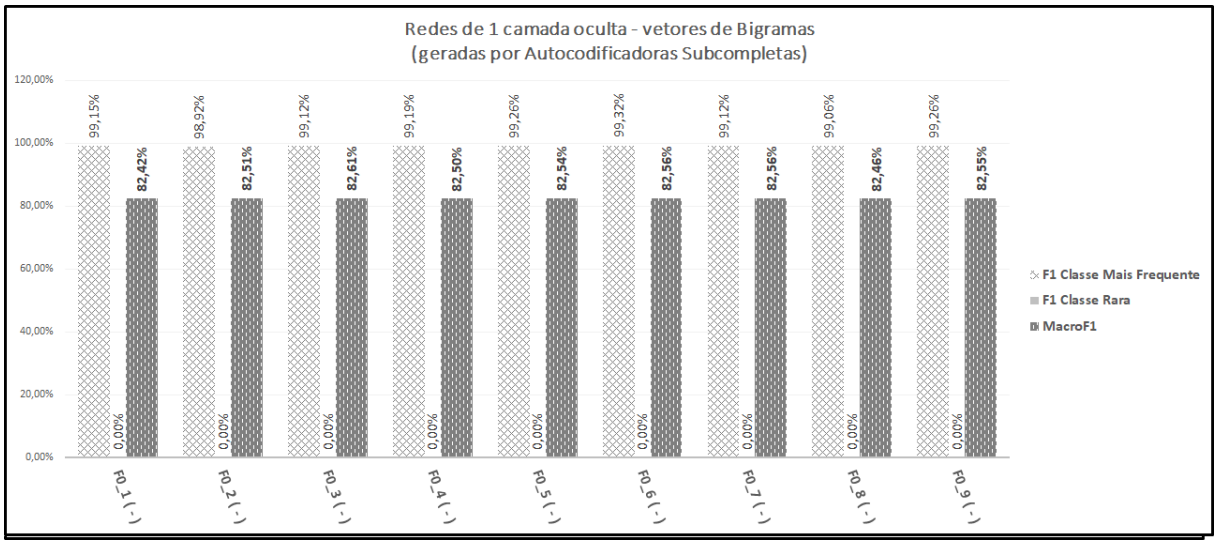

FIG. 6.12: Resultados de redes de 1 camada oculta para o conjunto de dados de Bigramas. Apenas modelos advindos de autocodificadoras subcompletas.

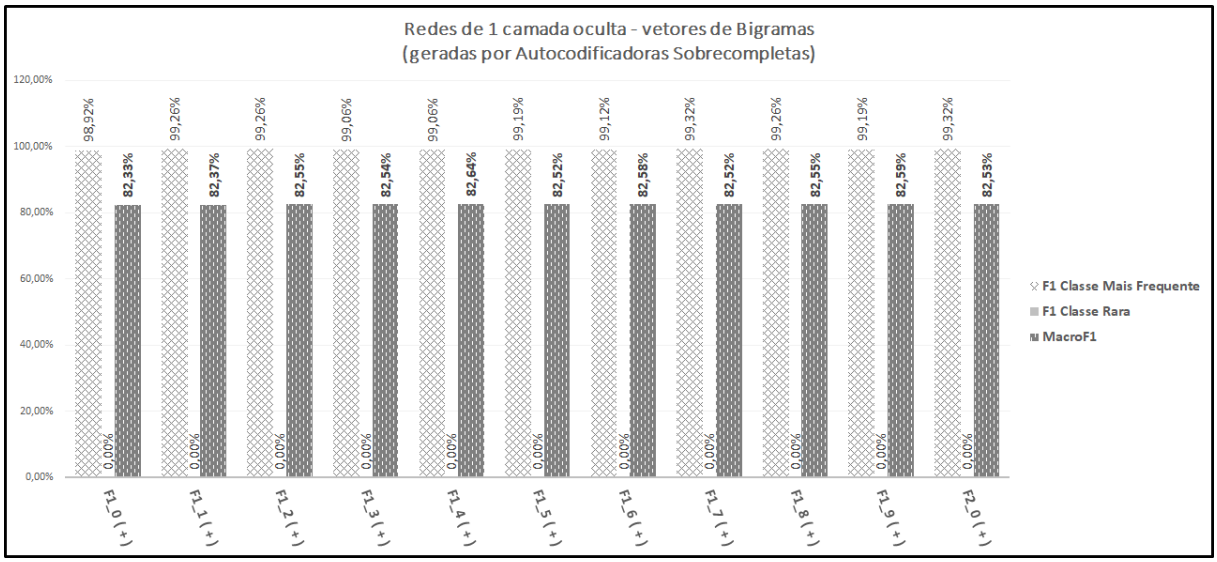

FIG. 6.13: Resultados de redes de 1 camada oculta para o conjunto de dados de Bigramas. Apenas modelos advindos de autocodificadoras sobrecompletas.

entre $42,86\%$ (menor valor, Rede 1) e $66,67\%$ (maior valor, Rede 5). $Macro_{F_1}$ variou de $89,98\%$ (menor valor, Rede 1) a $92,93\%$ (maior valor, Rede 5).

Nos modelos de **3 camadas ocultas** (gráfico na Figura 6.15), a Rede 1 se destaca por possuir o maior $Macro_{F_1}$ e F_1 para a classe rara, respectivamente $91,21\%$ e $52,63\%$. $Macro_{F_1}$ teve uma variação de $89,49\%$ (menor valor. Rede 5) a $91,21\%$ (maior valor, Rede 1). O F_1 da classe mais rara ficou entre $37,50\%$ (menor valor, Rede 5) e $52,63\%$ (maior valor, Rede 1), e o F_1 da classe mais frequente, $99,39\%$ em $99,46\%$.

Dos resultados para as redes de **4 camadas ocultas** (gráfico na Figura 6.16) destaca-

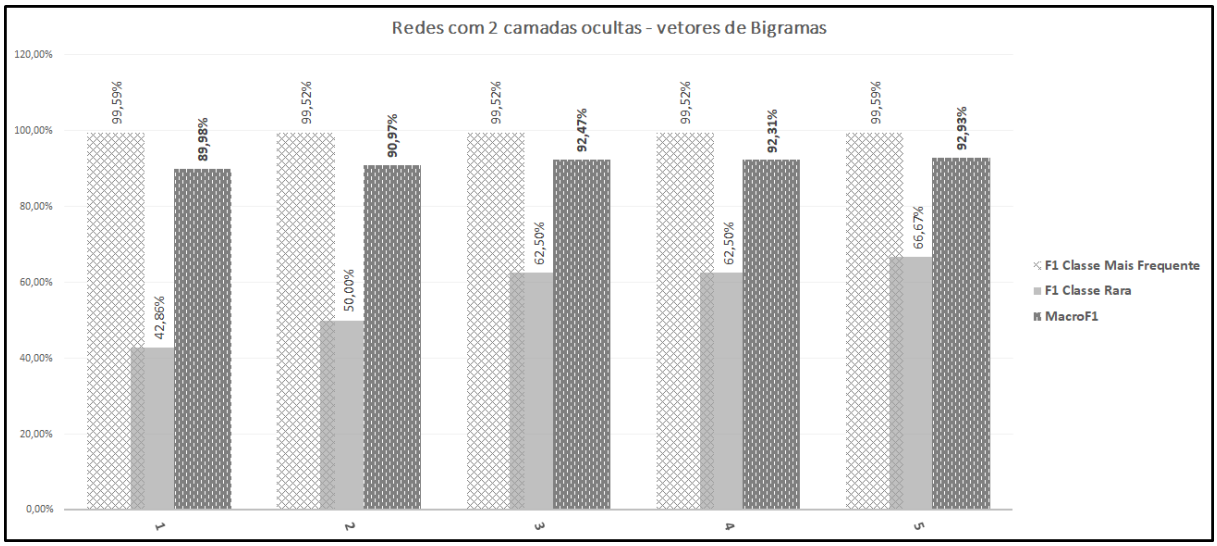

FIG. 6.14: Resultados de redes de 2 camadas ocultas para o conjunto de dados de bigramas.

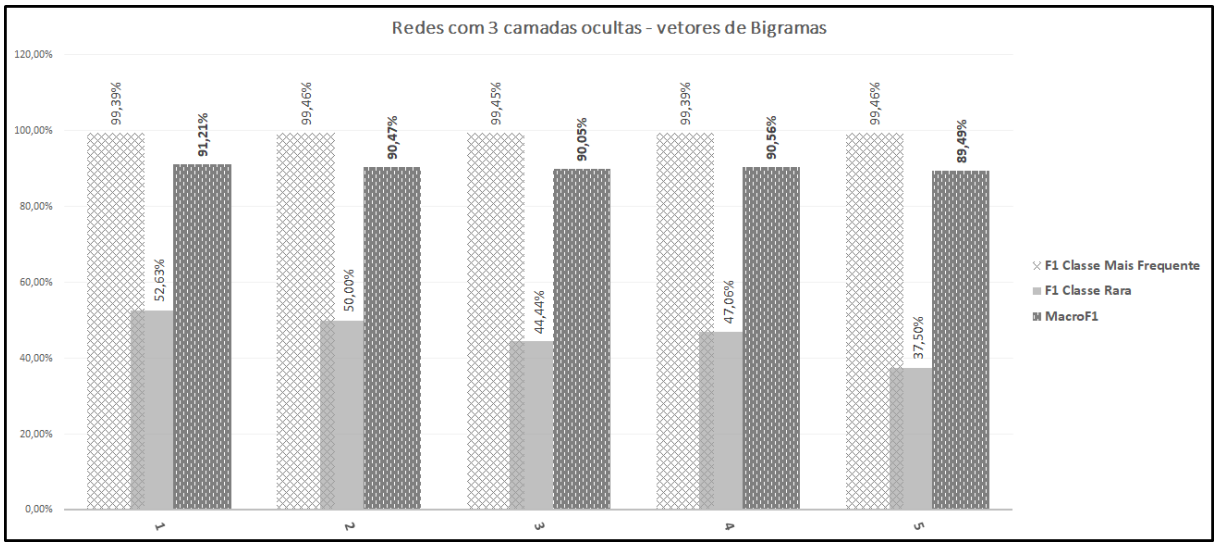

FIG. 6.15: Resultados de redes de 3 camadas ocultas para o conjunto de dados de bigramas.

se a Rede 4, que obteve os melhores resultados: $Macro_{F_1} = 93,14\%$, para a classe rara, $F_1 = 72,73\%$ e para a classe mais frequente, $F_1 = 99,52\%$. No geral, $Macro_{F_1}$ variou de $88,89\%$ (Rede 5) a $93,14\%$ (Rede 4). No caso da classe rara, o F_1 medido variou de $14,29\%$ (Rede 2) a $72,73\%$ (Rede 4). Para a classe mais frequente, todas as redes obtiveram a mesmo valor, com F_1 medindo $99,52\%$.

6.4.3 ANÁLISE DO MODELO FINAL

Podemos notar nos resultados dos 100 experimentos executados que a performance alcançada pelas redes treinadas com o conjunto de dados de bigramas, obtiveram em sua

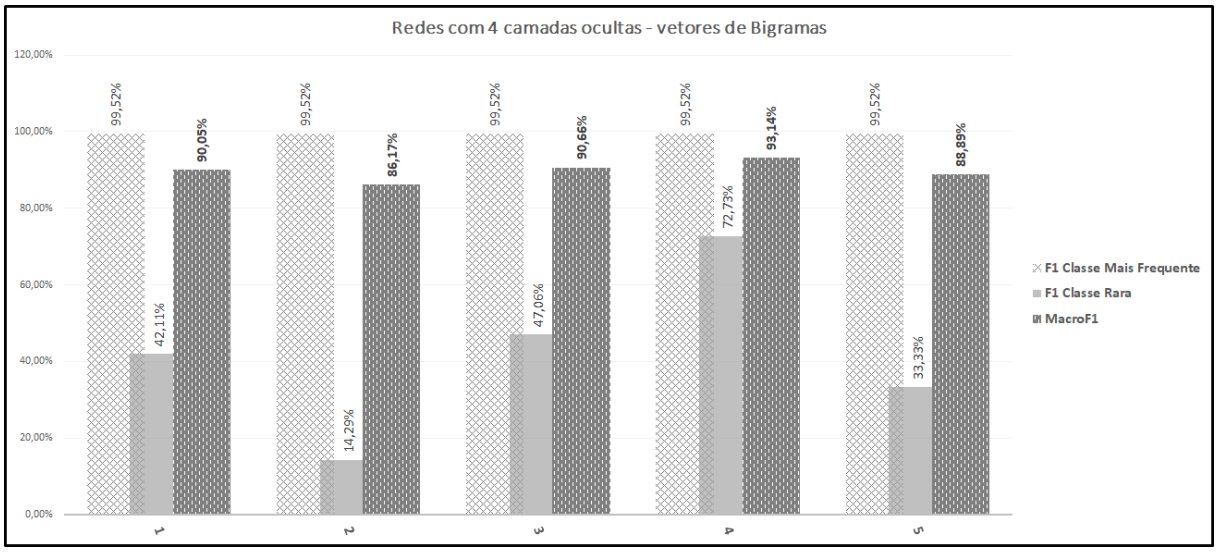

FIG. 6.16: Resultados de redes de 4 camadas ocultas para o conjunto de dados de bigramas.

maioria, um resultado superior aos dos modelos treinados no conjunto de dados de unigramas, o que já era de certa forma esperado, visto que o número maior de características contidas no conjunto de bigramas tende a impactar positivamente no desempenho do modelo. Selecionamos então, como o melhor classificador, o modelo que atingiu o maior valor de média $Macro_{F_1}$, que foi a quarta topologia gerada com 4 camadas ocultas treinada e validada no conjunto de dados de bigramas. Visualizando a matriz de confusão computada durante a fase de validação deste modelo (Tabela 6.3), podemos observar as predições realizadas pelo classificador baseadas em amostras não observadas anteriormente e se essa predição equivale ao rótulo de classe associado as amostras apresentadas. Podemos notar que apenas 90 amostras foram classificadas erroneamente, isso equivale a um pouco mais de 3% do total de amostras no conjunto de dados de validação. Partindo desta matriz de confusão, podemos extrair métricas para avaliar a performance do classificador.

Examinando a tabela 6.5, podemos analisar diversas métricas por classe. Excetuando a classe mais rara do conjunto de dados, todas as outras classes obtiveram a medida F_1 superior a 92%. Considerando isoladamente a classe mais rara, observamos que o modelo gerado conseguiu classificar grande parte das amostras raras a que foi apresentado, ou seja, mesmo com poucas amostras, o classificador conseguiu distinguir as amostras da classe rara de outras amostras, obtendo a maior medida F_1 para esta classe dentre todos os experimentos executados.

Ainda, por comparação, podemos observar a matriz de confusão da solução ganhadora do concurso promovido pela *Microsoft*, utilizando o mesmo conjunto de dados que foi

TAB. 6.3: Matriz de Confusão da Rede 4, Conjunto de Dados: Bigramas, melhor classificador.

	Prevista								
Real	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
1	**366**	4	0	0	0	1	0	15	0
2	4	**609**	0	1	2	0	0	1	3
3	0	0	**730**	0	0	5	1	0	0
4	0	3	1	**107**	0	6	0	2	0
5	0	0	0	0	**8**	2	1	0	0
6	0	2	0	2	0	**181**	0	3	0
7	0	1	0	0	0	1	**95**	1	2
8	4	2	0	1	0	4	0	**294**	2
9	0	6	0	0	1	3	0	3	**241**

TAB. 6.4: Matriz de confusão gerada a partir da execução no conjunto de dados de treinamento completo da abordagem ganhadora do concurso do Kaggle (KAGGLE, 2015)

	Prevista								
Real	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
1	**1541**	0	0	0	0	0	0	0	0
2	1	**2476**	0	0	0	1	0	0	0
3	0	0	**2942**	0	0	0	0	0	0
4	0	0	0	**475**	0	0	0	0	0
5	2	0	0	0	**39**	1	0	0	0
6	1	0	0	0	0	**750**	0	0	0
7	0	0	0	0	0	0	**398**	0	0
8	0	0	1	0	0	0	0	**1225**	2
9	0	1	0	0	0	0	0	5	**1007**

utilizado no início de nossa abordagem (Tabela 6.4). Partindo desta matriz de confusão, conseguimos calcular as mesmas métricas utilizadas em nossa abordagem (Tabela 6.6). Vemos que a abordagem ganhadora do concurso, apesar de ter executado sua solução em todas as 10868 amostras do conjunto de dados de treinamento do conjunto de dados original, tem performance superior ao classificador selecionado neste estudo, com diversas classes chegando muito próximas de 100% ou até mesmo em 100%. No caso da classe mais rara do conjunto de dados, a solução ganhadora do concurso obteve $F_1 = 96,30\%$, contra $F_1 = 72,73\%$ da abordagem deste estudo. Apenas por facilidade, calculamos o $Macro_{F_1}$ da solução ganhadora do concurso e obtivemos $99,46\%$. A solução do ganhador ainda é $6,32\%$ superior a nossa, comparado ao resultado de nossa abordagem, onde $Macro_{F_1} = 93,14\%$. Porém, devemos considerar que a abordagem ganhadora do concurso se baseou não apenas em conjuntos de n-gramas, mas também em diversas outras características extraídas do conjunto de dados, ao contrário da abordagem apresentada neste estudo, que

se baseia apenas na frequência dos OpCodes e operandos (KAGGLE, 2015).

Visto que os experimentos anteriores, no geral, nos mostraram uma melhora no valor do $Macro_{F_1}$ conforme aumentamos a quantidade de camadas ocultas, podemos esperar que classificadores melhores ainda possam aparecer. Dito isso, aliado aos resultados já encontrados em nosso trabalho inicial (PINTO; DUARTE, 2017), podemos concluir que existem fortes evidências de que a abordagem apresentada no estudo é efetiva para a classificação de amostras em famílias de *malware*.

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

	Métricas							
Classes	**VP**	**VN**	**FP**	**FN**	**Acurácia**	**Precisão**	**Revocação**	**F_1**
1	366	2327	8	20	98,97%	97,86%	94,82%	96,32%
2	609	2083	18	11	98,93%	97,13%	98,23%	97,67%
3	730	1984	1	6	99,74%	99,86%	99,18%	99,52%
4	107	2598	4	12	99,41%	96,40%	89,92%	93,04%
5	8	2707	3	3	99,78%	72,73%	72,73%	72,73%
6	181	2511	22	7	98,93%	89,16%	96,28%	92,58%
7	95	2619	2	5	99,74%	97,94%	95,00%	96,45%
8	294	2389	25	13	98,60%	92,16%	95,77%	93,93%
9	241	2460	7	13	99,27%	97,18%	94,88%	96,02%

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

	Métricas							
Classes	**VP**	**VN**	**FP**	**FN**	**Acurácia**	**Precisão**	**Revocação**	F_1
1	1541	9323	4	0	99,96%	99,74%	100,00%	99,87%
2	2476	8389	1	2	99,97%	99,96%	99,92%	99,94%
3	2942	7925	1	0	99,99%	99,97%	100,00%	99,98%
4	475	10393	0	0	100,00%	100,00%	100,00%	100,00%
5	39	10826	0	3	99,97%	100,00%	92,86%	96,30%
6	750	10115	2	1	99,97%	99,73%	99,87%	99,80%
7	398	10470	0	0	100,00%	100,00%	100,00%	100,00%
8	1225	9635	5	3	99,93%	99,59%	99,76%	99,67%
9	1007	9853	2	6	99,93%	99,80%	99,41%	99,60%

7 CONCLUSÃO

A atividade ilícita no ambiente cibernético cresce a cada ano, ameaçando áreas estratégicas e gerando prejuízos, algumas vezes incontáveis, para governos, empresas e usuários individuais ao redor do globo. Vimos que o desenvolvimento tecnológico com o passar do tempo trouxe não apenas a facilidade ou praticidade, mas também uma série de vulnerabilidades que são devidamente exploradas por diversas técnicas de ataque cibernético. Uma das armas utilizadas é o *malware*. O trabalho de analisar um programa suspeito e decidir se este possui instruções maliciosas fica a cargo de analistas de segurança, que apesar de utilizarem todo um ferramental automatizado que os auxiliem no trabalho, participam ativamente da atividade e normalmente estão em número menor em relação à quantidade de programas suspeitos a serem analisados. Daí a importância do estudo de técnicas que auxiliem sua detecção e classificação com um certo grau de confiança.

Durante a execução deste estudo, observamos o emprego de técnicas de Aprendizado de Máquina e também de Aprendizado Profundo no auxílio a análise de *malware*. As abordagens, utilizando ou não a análise estática, no geral baseiam-se fortemente em engenharia de características e conhecimento do domínio para conseguirem sucesso em seus objetivos. Observamos que, as que utilizaram OpCodes como características, as utilizaram calculando sua frequência por meio de diversas técnicas, porém nenhuma utilizou os operandos como possíveis características para a resolução do problema.

O objetivo geral definido para este trabalho foi o de avaliar a aplicabilidade de redes profundas como auxiliares a análise estática de *malware*, desprezando qualquer característica ou informação advinda de conhecimento prévio sobre cada família de *malware* pertencente as amostras utilizadas. Como objetivos específicos, foram definidos que deveria haver uma investigação do uso de contagens de OpCodes e operandos extraídos de *malware* desmontado no formato de Saco de Palavras, variando-se n-gramas e utilizando a ponderação por sua frequência. Outro foi a investigação de redes neurais Autocodificadoras profundas como descobridora de boas representações de dados advindas do código desmontado de *malware* e sua eficiência como pré-inicializadora de uma rede supervisionada para Classificação de *Malware*. E como último objetivo, a realização de uma análise de sensibilidade das redes neurais experimentadas, verificando seu desempenho e como reagem a alterações em seus parâmetros.

Para alcançar esses objetivos, foi criada uma abordagem que emprega o pré-treinamento

não-supervisionado, onde redes autocodificadoras profundas foram utilizadas para aprenderem características diretamente de dados não rotulados, para posteriormente inicializarem redes de múltiplas camadas para a classificação de *malware* em diversas famílias. Os dados, OpCodes e operandos no formato de Bytes em hexadecimal, originários de um conjunto de códigos desmontados de *malware*, foram transformados em uma representação de dados utilizando Saco de Palavras, para diversas combinações de termos – unigramas, bigramas e trigramas – utilizando contagens de OpCodes e Operandos, ponderado-os por TF-IDF. Para auxiliar a experimentação da abordagem, foi desenvolvida uma regra de formação das redes neurais, no intuito de cobrir uma ampla variedade de topologias. Para avaliação da qualidade desses classificadores, foram selecionadas as métricas para a escolha do melhor classificador dentre os experimentados.

Foram então executados um total de 100 experimentos, utilizando os conjuntos de dados de unigramas e bigramas. O melhor classificador, treinado com o conjunto de dados de bigramas, atingiu uma boa performance, com $Macro_{F_1} = 93,14\%$, para a classe mais rara, o $F_1 = 72,73\%$ e para a classe mais frequente $F_1 = 99,52\%$, comprovando a eficácia da abordagem.

Com isso, podemos sumarizar as contribuições deste trabalho. A primeira contribuição foi a criação de uma abordagem para classificação de *malware* dentre suas famílias, utilizando redes autocodificadoras profundas pré-inicializando redes profundas com múltiplas camadas, sem a utilização de quaisquer características que possam identificar o *malware*, utilizando apenas a frequência de OpCodes e Operandos em uma representação do tipo Saco de Palavras. Por consequência da primeira contribuição, criou-se um conjunto de dados baseado no conjunto de dados de treinamento da Microsoft (MICROSOFT, 2015; RONEN et al., 2018), criado através da extração de OpCodes e Operandos no formato de *Bytes* em hexadecimal, para Unigramas, Bigramas e Trigramas, ponderados pelo TF-IDF. Este conjunto de dados será disponibilizado para *download*.

Como sugestões para trabalhos futuros, pode-se sugerir trabalhos que podem ser aplicados como extensão da abordagem definida neste estudo ou como uma alteração mais significativa.

A utilização de outras arquiteturas de redes profundas poderia ser explorada em outros trabalhos. Particularmente, Redes Recorrentes, como por exemplo *Long Short-Term Memory* podem ajudar a reconhecer sequências de execução de instruções que levem a detectar ou classificar o *malware*.

Outro trabalho possível seria ampliar o escopo da abordagem atual para utilizar o conjunto de dados de vetores de trigramas já gerado durante a fase experimental deste

trabalho. Com isso, seria possível averiguar se a utilização de um número maior de combinações de OpCodes e operandos poderia melhorar a performance dos classificadores.

Uma maneira de aprimorar a abordagem desenvolvida neste estudo seria aplicá-la para a tarefa de detecção de *malware*. Para isso, um novo conjunto de dados deveria ser construído, incluindo além das amostras de *malware*, amostras benignas – programas inócuos. Para validar se as redes profundas conseguem realmente aprender os padrões complexos do código malicioso, seria importante selecionar amostras benignas que estão no limiar de serem reconhecidos como "código malicioso", ou seja, que fossem facilmente reconhecidos por um sistema de detecção como falsos positivos.

Um outro trabalho poderia explorar a utilização de uma nova representação dos dados extraídos do código desmontado do *malware*. *Word Embeddings* são técnicas de modelagem de linguagem muito utilizadas em Processamento de Linguagem Natural, onde palavras, frases e até mesmo parágrafos inteiros são mapeados para vetores numéricos. Essa representação vetorial procura capturar o máximo de informações morfológicas e semânticas do texto, informações essas desprezadas quando empregam-se técnicas baseadas em contagens de palavras e suas frequências. Ao analisar o código-fonte do *malware*, a relação existente entre os opcodes e os operandos poderia auxiliar na identificação de padrões de código malicioso para tarefas de classificação ou detecção de *malware*. Algumas das técnicas mais utilizadas são a word2vec (MIKOLOV et al., 2013), doc2vec (LE; MIKOLOV, 2014) e a GloVe (PENNINGTON et al., 2014).

8 REFERÊNCIAS BIBLIOGRÁFICAS

MINISTÉRIO DA DEFESA . Livro Branco de Defesa Nacional. Disponível em: <http://www.defesa.gov.br/arquivos/2012/mes07/lbdn.pdf>. Acesso em: 24 nov. de 2016.

ABADI, M.; BARHAM, P.; CHEN, J.; CHEN, Z.; DAVIS, A.; DEAN, J.; DEVIN, M.; GHEMAWAT, S.; IRVING, G.; ISARD, M. ; OTHERS. Tensorflow: a system for large-scale machine learning.. In: OSDI, 12th., 2016. **Anais**... [S.l.: s.n.], 2016, p. 265–283.

ADLEMAN, L. M. An abstract theory of computer viruses. In: ADVANCES IN CRYPTOLOGY — CRYPTO' 88, 1., 1990. **Anais**... New York, NY: Springer New York, 1990, p. 354–374.

ANDRYCHOWICZ, M.; DENIL, M.; GOMEZ, S.; HOFFMAN, M. W.; PFAU, D.; SCHAUL, T.; SHILLINGFORD, B. ; DE FREITAS, N. Learning to learn by gradient descent by gradient descent. **Advances in Neural Information Processing Systems**, v. 1, p. 3981–3989, 2016.

ARP, D.; SPREITZENBARTH, M.; HUBNER, M.; GASCON, H. ; RIECK, K. Drebin: Effective and explainable detection of android malware in your pocket.. In: NDSS, 18th., 2014. **Anais**... [S.l.: s.n.], 2014. Disponível em: <http://www.internetsociety.org/doc/drebin-effective-and-explainable-detection-android-malware-your-pocket>. Acesso em: 08 dez. 2016.

ARTHUR L SAMUEL. Some studies in machine learning using the game of checkers. **IBM Journal of research and development**, v. 3, n. 3, p. 210–229, 1959.

BAYER, U.; MOSER, A.; KRUEGEL, C. ; KIRDA, E. Dynamic analysis of malicious code. **Journal in Computer Virology**, v. 2, n. 1, p. 67–77, 2006.

BENGIO, Y.; COURVILLE, A. ; VINCENT, P. Representation learning: A review and new perspectives. **IEEE transactions on pattern analysis and machine intelligence**, v. 35, n. 8, p. 1798–1828, 2013.

BENGIO, Y.; OTHERS. Learning deep architectures for ai. **Foundations and trends® in Machine Learning**, v. 2, n. 1, p. 1–127, 2009.

BILAR, D. Fingerprinting malicious code through statistical opcode analysis. In: ICGES'07: PROCEEDINGS OF THE 3RD INTERNATIONAL CONFERENCE ON GLOBAL E-SECURITY, 3rd., 2007. **Anais**... [S.l.: s.n.], 2007, p. –.

BRANCO, R. R.; BARBOSA, G. N. ; NETO, P. D. Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies. **Black Hat**, v. 1, 2012. Disponível em: <https://ubm.io/2Ofdvtn>. Acesso em: 04 Ago. 2018.

CANI, A.; GAUDESI, M.; SANCHEZ, E.; SQUILLERO, G. ; TONDA, A. Towards automated malware creation: Code generation and code integration. In: PROCEEDINGS OF THE 29TH ANNUAL ACM SYMPOSIUM ON APPLIED COMPUTING, 29th., SAC '14, 4., 2014. **Anais**... [S.l.]: ACM, 2014, p. 157–160.

CHOLLET, FRANÇOIS AND OTHERS. Keras. Disponível em: <https://keras.io>. Acesso em: 10 Jul. 2018.

CHRISTODORESCU, M.; JHA, S. Static analysis of executables to detect malicious patterns. In: PROCEEDINGS OF THE 12TH CONFERENCE ON USENIX SECURITY SYMPOSIUM - VOLUME 12, 12th., SSYM'03, 1., 2003. **Anais**... [S.l.: s.n.], 2003, p. 12–12. Disponível em: <http://dl.acm.org/citation.cfm?id=1251353.1251365>. Acesso em: 31 Jul 2018.

COHEN, F. Computer viruses: theory and experiments. **Computers & security**, v. 6, n. 1, p. 22–35, 1987.

DAHL, G. E.; STOKES, J. W.; DENG, L. ; YU, D. Large-scale malware classification using random projections and neural networks. In: 2013 IEEE INTERNATIONAL CONFERENCE ON ACOUSTICS, SPEECH AND SIGNAL PROCESSING, 38th., 2013. **Anais**... [S.l.: s.n.], 2013, p. 3422–3426.

DAMSHENAS, M.; DEHGHANTANHA, A. ; MAHMOUD, R. A survey on malware propagation, analysis, and detection. **International Journal of Cyber-Security and Digital Forensics (IJCSDF)**, v. 2, n. 4, p. 10–29, 2013.

DEMUTH, H. B.; BEALE, M. H.; DE JESS, O. ; HAGAN, M. T. **Neural Network Design**. 2nd. ed. USA: Martin Hagan, 2014. ISBN 0971732116, 9780971732117.

DENG, L.; YU, D. ; OTHERS. Deep learning: methods and applications. **Foundations and Trends® in Signal Processing**, v. 7, n. 3–4, p. 197–387, 2014.

DOERSCH, CARL. Tutorial on variational autoencoders. Disponível em: <https://arxiv.org/abs/1606.05908>. Acesso em: 10 Jul. 2018.

DOMINGOS, P. A few useful things to know about machine learning. **Commun. ACM**, v. 55, n. 10, p. 78–87, 2012.

EGELE, M.; SCHOLTE, T.; KIRDA, E. ; KRUEGEL, C. A survey on automated dynamic malware-analysis techniques and tools. **ACM Comput. Surv.**, v. 44, n. 2, p. 6:1–6:42, 2008. Disponível em: <http://doi.acm.org/10.1145/2089125.2089126>. Acesso em: 12 Mai. 2018.

ENISA. Reference Incident Classification Taxonomy: Taskforce status and Way forward. Disponível em: <https://www.enisa.europa.eu/publications/reference-incident-classification-taxonomy>. Acesso em: 30 mar. de 2018.

ERHAN, D.; BENGIO, Y.; COURVILLE, A.; MANZAGOL, P.-A.; VINCENT, P. ; BENGIO, S. Why does unsupervised pre-training help deep learning?. **Journal of Machine Learning Research**, v. 11, n. Feb, p. 625–660, 2010.

FACELI, K.; LORENA, A. C. G. J. . C. A. **Inteligência Artificial: Uma abordagem de aprendizado de máquina**. [S.l.]: LTC, 2011.

FUKUSHIMA, K.; MIYAKE, S. Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In: COMPETITION AND COOPERATION IN NEURAL NETS, –., 1982. **Anais**... Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, p. 267–285.

GLOROT, X.; BORDES, A. ; BENGIO, Y. Deep sparse rectifier neural networks. In: PROCEEDINGS OF THE FOURTEENTH INTERNATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE AND STATISTICS, 14th., 2011. **Anais**... [S.l.: s.n.], 2011, p. 315–323.

GOODFELLOW, I.; BENGIO, Y. ; COURVILLE, A. **Deep Learning**. 1. ed. [S.l.]: MIT Press, 2016. 800 p.

GRAVES, A. **Supervised Sequence Labelling with Recurrent Neural Networks**. [S.l.]: Springer, 2012. 1-131 p. ISBN 978-3-642-24796-5.

GUO, Y.; LIU, Y.; OERLEMANS, A.; LAO, S.; WU, S. ; LEW, M. S. Deep learning for visual understanding: A review. **Neurocomputing**, v. 187, p. 27–48, 2016.

HAJIGHORBANI, M.; HASHEMI, S. R.; MINAEI-BIDGOLI, B. ; SAFARI, S. A review of some semi-supervised learning methods. In: IEEE-2016, FIRST INTERNATIONAL CONFERENCE ON NEW RESEARCH ACHIEVEMENTS IN ELECTRICAL AND COMPUTER ENGINEERING, 1st., 2016. **Anais**... [S.l.: s.n.], 2016, p. 250–259.

HARDY, W.; CHEN, L.; HOU, S.; YE, Y. ; LI, X. Dl4md: A deep learning framework for intelligent malware detection. In: PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON DATA MINING (DMIN), 12th., 2016. **Anais**... [S.l.: s.n.], 2016, p. 61.

HAYKIN, S. **Neural networks: A comprehensive foundation**. [S.l.: s.n.], 2004.

HAYKIN, S. S.; HAYKIN, S. S.; HAYKIN, S. S. ; HAYKIN, S. S. **Neural networks and learning machines**. [S.l.]: Pearson Upper Saddle River, NJ, USA:, 2009.

HEX-RAYS. IDA Pro Disassembler. Disponível em: <https://www.hexrays.com/products/ida/index.shtml>. Acesso em: 02 ago. de 2017.

HINTON, G. E.; SALAKHUTDINOV, R. R. Reducing the dimensionality of data with neural networks. **science**, v. 313, n. 5786, p. 504–507, 2006.

HUBEL, D. H.; WIESEL, T. N. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. **The Journal of physiology**, v. 160, n. 1, p. 106–154, 1962.

JANG-JACCARD, J.; NEPAL, S. A survey of emerging threats in cybersecurity. **Journal of Computer and System Sciences**, v. 80, n. 5, p. 973–993, 2014.

KAGGLE. Microsoft Malware Winners' Interview: 1st place, NO to overfitting. Disponível em: <http://blog.kaggle.com/2015/05/26/microsoft-malware-winners-interview-1st-plaece-no-to-overfitting>. Acesso em: 01 Ago. 2018.

ANDREJ KARPATHY. CS231n: Convolutional Neural Networks for Visual Recognition. Disponível em: <http://cs231n.github.io/>. Acesso em: 21 ago. de 2018.

KINGMA, DIEDERIK P AND WELLING, MAX. Auto-encoding variational bayes. Disponível em: <https://arxiv.org/abs/1312.6114>. Acesso em: 1 Jul. 2018.

LE, Q.; MIKOLOV, T. Distributed representations of sentences and documents. In: INTERNATIONAL CONFERENCE ON MACHINE LEARNING, 31st., 2014. **Anais**... [S.l.: s.n.], 2014, p. 1188–1196.

LECUN, Y.; BENGIO, Y. ; HINTON, G. Deep learning. **Nature**, v. 521, n. 7553, p. 436–444, 2015.

LECUN, Y.; BOSER, B.; DENKER, J. S.; HENDERSON, D.; HOWARD, R. E.; HUBBARD, W. ; JACKEL, L. D. Backpropagation applied to handwritten zip code recognition. **Neural computation**, v. 1, n. 4, p. 541–551, 1989.

LECUN, Y.; BOTTOU, L.; BENGIO, Y. ; HAFFNER, P. Gradient-based learning applied to document recognition. **Proceedings of the IEEE**, v. 86, n. 11, p. 2278–2324, 1998.

MANGIALARDO, R. J.; DUARTE, J. C. Integrating static and dynamic malware analysis using machine learning. **IEEE Latin America Transactions**, v. 13, n. 9, p. 3080–3087, 2015.

MANGIALARDO, R. J. **Integrando as Análises Estática e Dinâmica na Identificação de Malwares Utilizando Aprendizado de Máquina**. 2015. 105 f. Dissertação (Mestrado em Sistemas e Computação) – Instituto Militar de Engenharia, Rio de Janeiro, 2015.

MANNING, C. D.; RAGHAVAN, P. ; SCHÜTZE, H. Scoring, term weighting and the vector space model. **Introduction to information retrieval**, v. 100, p. 2–4, 2008.

MANNING, C. D.; SCHÜTZE, H. ; RAGHAVAN, P. **Introduction to information retrieval**. [S.l.]: Cambridge University Press, 2008.

MATHUR, K.; HIRANWAL, S. A survey on techniques in detection and analyzing malware executables. **International Journal of Advanced Research in Computer Science and Software Engineering**, v. 3, n. 4, p. 422–428, 2013.

MCAFEE. Net Losses: Estimating the Global Cost of Cybercrime. Intel Security (McAfee). Disponível em: <http://www.mcafee.com/us/resources/reports/rp-economic-impact-cybercrime2.pdf>. Acesso em: 25 nov. de 2016.

MCAFEE. Previsões do McAfee Labs sobre ameaças em 2016. Disponível em: <http://www.mcafee.com/br/resources/reports/rp-threats-predictions-2016.pdf>. Acesso em: 25 nov. de 2016.

MCCULLOCH, W. S.; PITTS, W. A logical calculus of the ideas immanent in nervous activity. **The bulletin of mathematical biophysics**, v. 5, n. 4, p. 115–133, 1943. Disponível em: <https://doi.org/10.1007/BF02478259>. Acesso em: 10 Set. 2017.

MCGRAW, G.; MORRISETT, G. Attacking malicious code: A report to the infosec research council. **IEEE software**, v. 17, n. 5, p. 33, 2000.

MICROSOFT. Microsoft Malware Classification Challenge (BIG 2015). Disponível em: <https://www.kaggle.com/c/malware-classification>. Acesso em: 02 ago. de 2017.

MIKOLOV, T.; SUTSKEVER, I.; CHEN, K.; CORRADO, G. S. ; DEAN, J. Distributed representations of words and phrases and their compositionality. In: ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS, 26., 2013. **Anais**... [S.l.: s.n.], 2013, p. 3111–3119.

MITCHELL, T. M. **Machine Learning**. 1st. ed. New York, NY, USA: McGraw-Hill, Inc., 1997. ISBN 0070428077, 9780070428072.

MONARD, M. C.; BARANAUSKAS, J. A. **Conceitos Sobre Aprendizado de Máquina**. 1. ed. Barueri-SP: Manole Ltda, 2003. 89–114 p. ISBN 85-204-168.

MURPHY, K. P. **Machine Learning: A Probabilistic Perspective**. [S.l.]: The MIT Press, 2012. ISBN 0262018020, 9780262018029.

THE COUNCIL OF ECONOMIC ADVISERS. The Cost of Malicious Cyber Activity to the U.S. Economy. Disponível em: <https://www.whitehouse.gov/wp-content/uploads/2018/03/The-Cost-of-Malicious-Cyber-Activity-to-the-U.S.-Economy.pdf>. Acesso em: 16 mai. de 2018.

PEDREGOSA, F.; VAROQUAUX, G.; GRAMFORT, A.; MICHEL, V.; THIRION, B.; GRISEL, O.; BLONDEL, M.; PRETTENHOFER, P.; WEISS, R.; DUBOURG, V.; VANDERPLAS, J.; PASSOS, A.; COURNAPEAU, D.; BRUCHER, M.; PERROT, M. ; DUCHESNAY, E. Scikit-learn: Machine learning in Python. **Journal of Machine Learning Research**, v. 12, p. 2825–2830, 2011.

PENNINGTON, J.; SOCHER, R. ; MANNING, C. D. Glove: Global vectors for word representation. In: EMPIRICAL METHODS IN NATURAL LANGUAGE PROCESSING (EMNLP), 14., 2014. **Anais**... [S.l.: s.n.], 2014, p. 1532–1543. Disponível em: <http://www.aclweb.org/anthology/D14-1162>. Acesso em: 1 Ago. 2018.

PINTO, D. R.; DUARTE, J. Static analysis on disassembled files: A deep learning approach to malware classification. In: SBSEG 2017 (XVII SIMPÓSIO BRASILEIRO EM SEGURANÇA DA INFORMAÇÃO E DE SISTEMAS COMPUTACIONAIS), 17., 2017. **Anais**... [S.l.: s.n.], 2017, p. 471–478.

MISP OPEN SOURCE THREAT INTELLIGENCE PLATFORM. MISP Taxonomies and classification as machine tags. Disponível em: <https://www.misp-project.org/taxonomies.pdf>. Acesso em: 31 mar. 2018.

PRAKASH, V. J.; NITHYA, L. M. A survey on semi-supervised learning techniques. **CoRR**, v. abs/1402.4645, 2014. Disponível em: <http://arxiv.org/abs/1402.4645>. Acesso em: 30 Jul. 2018.

PRESIDÊNCIA DA REPÚBLICA. Livro Verde: Segurança Cibernética no Brasil. Disponível em: <http://dsic.planalto.gov.br>. Acesso em: 24 nov. de 2016.

REZENDE, D. J.; MOHAMED, S. ; WIERSTRA, D. Stochastic backpropagation and approximate inference in deep generative models. In: PROCEEDINGS OF THE 31ST INTERNATIONAL CONFERENCE ON MACHINE LEARNING, 31st., 2014. **Anais**... [S.l.: s.n.], 2014, p. 1278–1286.

RIFAI, S.; VINCENT, P.; MULLER, X.; GLOROT, X. ; BENGIO, Y. Contractive auto-encoders: Explicit invariance during feature extraction. In: PROCEEDINGS OF THE 28TH INTERNATIONAL CONFERENCE ON INTERNATIONAL CONFERENCE ON MACHINE LEARNING, 28th., 2011. **Anais**... [S.l.: s.n.], 2011, p. 833–840.

RONEN, R.; RADU, M.; FEUERSTEIN, C.; YOM-TOV, E. ; AHMADI, M. Microsoft malware classification challenge. **CoRR**, v. abs/1802.10135, 2018. Disponível em: <http://arxiv.org/abs/1802.10135>. Acesso em: 3 Set. 2018.

ROSENBLATT, F. The perceptron: A probabilistic model for information storage and organization in the brain.. **Psychological review**, v. 65, n. 6, p. 386, 1958.

SEBASTIAN RUDER. An overview of gradient descent optimization algorithms. Disponível em: <http://arxiv.org/abs/1609.04747>. Acesso em: 03 Set. 2018.

SADARANGANI, A.; JIVANI, A. A survey of semi-supervised learning. **INTERNATIONAL JOURNAL OF ENGINEERING SCIENCES & RESEARCH TECHNOLOGY A SURVEY OF SEMI-SUPERVISED LEARNING**, v. 10, 2016. Disponível em: <https://bit.ly/2vKFbP1>. Acesso em: 25 Jul. 2018.

SANTOS, I.; BREZO, F.; UGARTE-PEDRERO, X. ; BRINGAS, P. G. Opcode sequences as representation of executables for data-mining-based unknown malware detection. **Information Sciences**, v. 231, p. 64–82, 2013.

SCHMIDHUBER, J. Deep learning in neural networks: An overview. **Neural Networks**, v. 61, p. 85–117, 2015.

SHABTAI, A.; MOSKOVITCH, R.; FEHER, C.; DOLEV, S. ; ELOVICI, Y. Detecting unknown malicious code by applying classification techniques on opcode patterns. **Security Informatics**, v. 1, n. 1, p. 1, 2012.

SIKORSKI, M.; HONIG, A. **Practical malware analysis: the hands-on guide to dissecting malicious software**. 1. ed. [S.l.]: no starch press, 2012. 800 p.

SOKOLOVA, M.; LAPALME, G. A systematic analysis of performance measures for classification tasks. **Information Processing & Management**, v. 45, n. 4, p. 427–437, 2009.

SYMANTEC. Internet Security Threat Report. Disponível em: <https://www.symantec.com/content/dam/symantec/docs/reports/istr-21-2016-en.pdf>. Acesso em: 25 nov. de 2016.

SYMANTEC. Internet Security Threat Report. Disponível em: <https://www.symantec.com/content/dam/symantec/docs/reports/istr-23-2018-en.pdf>. Acesso em: 15 mai. de 2018.

MARK M. TEHRANIPOOR AND UJJWAL GUIN AND SWARUP BHUNIA. Invasion of the Hardware Snatchers: Cloned Electronics Pollute the Market. Disponível em: <https://spectrum.ieee.org/computing/hardware/invasion-of-the-hardware-snatchers-cloned-electronics-pollute-the-market>. Acesso em: 03 abr. de 2018.

VERISON. 2016 Data Breach Investigations Report. Disponível em: <http://www.verizonenterprise.com/verizon-insights-lab/dbir/2016/>. Acesso em: 25 nov. de 2016.

VERIZON. Data Breach Digest. Scenarios from the field. Disponível em: <https://vz.to/2LQmi8m>. Acesso em: 18 mai. de 2018.

VINCENT, P.; LAROCHELLE, H.; LAJOIE, I.; BENGIO, Y. ; MANZAGOL, P.-A. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. **Journal of Machine Learning Research**, v. 11, n. Dec, p. 3371–3408, 2010.

YOU, I.; YIM, K. Malware obfuscation techniques: A brief survey. In: 2010 INTERNATIONAL CONFERENCE ON BROADBAND, WIRELESS COMPUTING, COMMUNICATION AND APPLICATIONS, 5., 2010. **Anais**... [S.l.: s.n.], 2010, p. 297–300.

YUAN, Z.; LU, Y.; WANG, Z. ; XUE, Y. Droid-sec: Deep learning in android malware detection. In: ACM SIGCOMM COMPUTER COMMUNICATION REVIEW, 14th., 2014. **Anais**... [S.l.: s.n.], 2014, p. 371–372.

YUXIN, D.; SIYI, Z. Malware detection based on deep learning algorithm. **Neural Computing and Applications**, v. 28, p. 1–12, 2017.

ZELTSER, LENNY. How Malware Defends Itself Using TLS Callback Functions. Disponível em: <https://isc.sans.edu/diary/How+Malware+Defends+Itself+Using+TLS+Callback+Functions/66
Acesso em: 02 mai. de 2018.

9 APÊNDICES

APÊNDICE 1: CONFIGURAÇÕES DAS REDES: UNIGRAMAS

TAB. 9.1: Experimentos com vetores de unigramas: Configurações das Redes Classificadoras com 1 camada oculta

Experimento	**Autocodificadoras**	**Classificadoras**
F0_1	96-9-96	96-9-9
F0_2	96-19-96	96-19-9
F0_3	96-28-96	96-28-9
F0_4	96-38-96	96-38-9
F0_5	96-48-96	96-48-9
F0_6	96-57-96	96-57-9
F0_7	96-67-96	96-67-9
F0_8	96-76-96	96-76-9
F0_9	96-86-96	96-86-9
F1_0	96-96-96	96-96-9
F1_1	96-105-96	96-105-9
F1_2	96-115-96	96-115-9
F1_3	96-124-96	96-124-9
F1_4	96-134-96	96-134-9
F1_5	96-144-96	96-144-9
F1_6	96-153-96	96-153-9
F1_7	96-163-96	96-163-9
F1_8	96-172-96	96-172-9
F1_9	96-182-96	96-182-9
F2_0	96-192-96	96-192-9

TAB. 9.2: Experimentos com vetores de unigramas: Configurações das Redes Classificadoras com 2 camadas ocultas

Experimento	**Autocodificadoras**	**Classificadoras**
01	96-144-130-144-96	96-144-130-9
02	96-134-121-134-96	96-134-121-9
02	96-19-18-19-96	96-19-18-9
04	96-163-148-163-96	96-163-148-9
05	96-192-174-192-96	96-192-174-9

TAB. 9.3: Experimentos com vetores de unigramas: Configurações das Redes Classificadoras com 3 camadas ocultas

Experimento	**Autocodificadoras**	**Classificadoras**
01	96-144-130-117-130-144-96	96-144-130-117-9
02	96-134-121-109-121-134-96	96-134-121-109-9
03	96-19-18-17-18-19-96	96-19-18-17-9
04	96-163-148-132-148-163-96	96-163-148-132-9
05	96-192-174-155-174-192-96	96-192-174-155-9

TAB. 9.4: Experimentos com vetores de unigramas: Configurações das Redes Classificadoras com 4 camadas ocultas

Experimento	Autocodificadoras	Classificadoras
01	96-144-130-117-103-117-130-144-96	96-144-130-117-103-9
02	96-134-121-109-96-109-121-134-96	96-134-121-109-96-9
03	96-19-18-17-16-17-18-19-96	96-19-18-17-16-9
04	96-163-148-132-117-132-148-163-96	96-163-148-132-117-9
05	96-192-174-155-137-155-174-192-96	96-192-174-155-137-9

TAB. 9.5: Experimentos com vetores de unigramas: Configurações das Redes Classificadoras com 5 camadas ocultas

Experimento	Autocodificadoras	Classificadoras
01	96-144-130-117-103-90-103-117-130-144-96	96-144-130-117-103-90-9
02	96-134-121-109-96-84-96-109-121-134-96	96-134-121-109-96-84-9
03	96-19-18-17-16-15-16-17-18-19-96	96-19-18-17-16-15-9
04	96-163-148-132-117-101-117-132-148-163-96	96-163-148-132-117-101-9
05	96-192-174-155-137-119-137-155-174-192-96	96-192-174-155-137-119-9

TAB. 9.6: Experimentos com vetores de unigramas: Configurações das Redes Classificadoras com 6 camadas ocultas

Experimento	Autocodificadoras	Classificadoras
01	96-144-130-117-103-90-76-90-103-117-130-144-96	96-144-130-117-103-90-76-9
02	96-134-121-109-96-84-71-84-96-109-121-134-96	96-134-121-109-96-84-71-9
03	96-19-18-17-16-15-14-15-16-17-18-19-96	96-19-18-17-16-15-14-9
04	96-163-148-132-117-101-86-101-117-132-148-163-96	96-163-148-132-117-101-86-9
05	96-192-174-155-137-119-100-119-137-155-174-192-96	96-192-174-155-137-119-100-9

TAB. 9.7: Experimentos com vetores de unigramas: Configurações das Redes Classificadoras com 7 camadas ocultas

Experimento	Autocodificadoras	Classificadoras
01	96-144-130-117-103-90-76-63-76-90-103-117-130-144-96	96-144-130-117-103-90-76-63-9
02	96-134-121-109-96-84-71-59-71-84-96-109-121-134-96	96-134-121-109-96-84-71-59-9
03	96-19-18-17-16-15-14-13-14-15-16-17-18-19-96	96-19-18-17-16-15-14-13-9
04	96-163-148-132-117-101-86-71-86-101-117-132-148-163-96	96-163-148-132-117-101-86-71-9
05	96-192-174-155-137-119-100-82-100-119-137-155-174-192-96	96-192-174-155-137-119-100-82-9

TAB. 9.8: Experimentos com vetores de unigramas: Configurações das Redes Classificadoras com 8 camadas ocultas

Experimento	**Autocodificadoras**	**Classificadoras**
01	96-144-130-117-103-90-76-63-49-63-76-90-103-117-130-144-96	96-144-130-117-103-90-76-63-49-9
02	96-134-121-109-96-84-71-59-46-59-71-84-96-109-121-134-96	96-134-121-109-96-84-71-59-46-9
03	96-19-18-17-16-15-14-13-12-13-14-15-16-17-18-19-96	96-19-18-17-16-15-14-13-12-9
04	96-163-148-132-117-101-86-71-55-71-86-101-117-132-148-163-96	96-163-148-132-117-101-86-71-55-9
05	96-192-174-155-137-119-100-82-64-82-100-119-137-155-174-192-96	96-192-174-155-137-119-100-82-64-9

TAB. 9.9: Experimentos com vetores de unigramas: Configurações das Redes Classificadoras com 9 camadas ocultas

Experimento	**Autocodificadoras**	**Classificadoras**
01	96-144-130-117-103-90-76-63-49-36-49-63-76-90-103-117-130-144-96	96-144-130-117-103-90-76-63-49-36-9
02	96-134-121-109-96-84-71-59-46-34-46-59-71-84-96-109-121-134-96	96-134-121-109-96-84-71-59-46-34-9
03	96-19-18-17-16-15-14-13-12-11-12-13-14-15-16-17-18-19-96	96-19-18-17-16-15-14-13-12-11-9
04	96-163-148-132-117-101-86-71-55-40-55-71-86-101-117-132-148-163-96	96-163-148-132-117-101-86-71-55-40-9
05	96-192-174-155-137-119-100-82-64-46-64-82-100-119-137-155-174-192-96	96-192-174-155-137-119-100-82-64-46-9

TAB. 9.10: Experimentos com vetores de unigramas: Configurações das Redes Classificadoras com 10 camadas ocultas

Experimento	**Autocodificadoras**	**Classificadoras**
01	96-144-130-117-103-90-76-63-49-36-22-36-49-63-76-90-103-117-130-144-96	96-144-130-117-103-90-76-63-49-36-22-9
02	96-134-121-109-96-84-71-59-46-34-21-34-46-59-71-84-96-109-121-134-96	96-134-121-109-96-84-71-59-46-34-21-9
03	96-19-18-17-16-15-14-13-12-11-10-11-12-13-14-15-16-17-18-19-96	96-19-18-17-16-15-14-13-12-11-10-9
04	96-163-148-132-117-101-86-71-55-40-24-40-55-71-86-101-117-132-148-163-96	96-163-148-132-117-101-86-71-55-40-24-9
05	96-192-174-155-137-119-100-82-64-46-27-46-64-82-100-119-137-155-174-192-96	96-192-174-155-137-119-100-82-64-46-27-9

APÊNDICE 2: CONFIGURAÇÕES DAS REDES: BIGRAMAS

TAB. 9.11: Experimentos com vetores de bigramas: Configurações das Redes Classificadoras com 1 camada oculta

Experimento	Autocodificadoras	Classificadoras
F0_1	9216-922-9216	9216-922-9
F0_2	9216-1843-9216	9216-1843-9
F0_3	9216-2765-9216	9216-2765-9
F0_4	9216-3686-9216	9216-3686-9
F0_5	9216-4608-9216	9216-4608-9
F0_6	9216-5530-9216	9216-5530-9
F0_7	9216-6451-9216	9216-6451-9
F0_8	9216-7373-9216	9216-7373-9
F0_9	9216-8294-9216	9216-8294-9
F1_0	9216-9216-9216	9216-9216-9
F1_1	9216-10138-9216	9216-10138-9
F1_2	9216-11059-9216	9216-11059-9
F1_3	9216-11981-9216	9216-11981-9
F1_4	9216-12902-9216	9216-12902-9
F1_5	9216-13824-9216	9216-13824-9
F1_6	9216-14746-9216	9216-14746-9
F1_7	9216-15667-9216	9216-15667-9
F1_8	9216-16589-9216	9216-16589-9
F1_9	9216-17510-9216	9216-17510-9
F2_0	9216-18432-9216	9216-18432-9

TAB. 9.12: Experimentos com vetores de bigramas: Configurações das Redes Classificadoras com 2 camadas ocultas

Experimento	Autocodificadoras	Classificadoras
01	9216-9216-8295-9216-9216	9216-9216-8295-9
02	9216-14746-13272-14746-9216	9216-14746-13272-9
03	9216-15667-14101-15667-9216	9216-15667-14101-9
04	9216-16589-14931-16589-9216	9216-16589-14931-9
05	9216-18432-16590-18432-9216	9216-18432-16590-9

TAB. 9.13: Experimentos com vetores de bigramas: Configurações das Redes Classificadoras com 3 camadas ocultas

Experimento	Autocodificadoras	Classificadoras
01	9216-9216-8295-9216-9216	9216-9216-8295-9
02	9216-14746-13272-14746-9216	9216-14746-13272-9
03	9216-15667-14101-15667-9216	9216-15667-14101-9
04	9216-16589-14931-16589-9216	9216-16589-14931-9
05	9216-18432-16590-18432-9216	9216-18432-16590-9

TAB. 9.14: Experimentos com vetores de bigramas: Configurações das Redes Classificadoras com 4 camadas ocultas

Experimento	Autocodificadoras	Classificadoras
01	9216-9216-8295-9216-9216	9216-9216-8295-9
02	9216-14746-13272-14746-9216	9216-14746-13272-9
03	9216-15667-14101-15667-9216	9216-15667-14101-9
04	9216-16589-14931-16589-9216	9216-16589-14931-9
05	9216-18432-16590-18432-9216	9216-18432-16590-9

TAB. 9.15: Resultado dos Experimentos com vetores de unigramas para redes classificadoras com 1 camada oculta

Experimento	**Macro$_{\mathbf{F_1}}$**
F0_1	0,743641
F0_2	0,763707
F0_3	0,748919
F0_4	0,761061
F0_5	0,750666
F0_6	0,744856
F0_7	0,754246
F0_8	0,753150
F0_9	0,710870
F1_0	0,753471
F1_1	0,758557
F1_2	0,754860
F1_3	0,766602
F1_4	0,767564
F1_5	0,768340
F1_6	0,755103
F1_7	0,763047
F1_8	0,743332
F1_9	0,749563
F2_0	0,761248

TAB. 9.16: Resultado geral das redes classificadoras até 10 camadas ocultas, utilizando o conjunto de dados de vetores de unigramas

Experimento	**Qtd. Camadas Ocultas (Classificadora)**	**Macro$_{F_1}$**
01	2	0,826192
02	2	0,824531
03	2	0,769776
04	2	0,843021
05	2	0,836504
01	3	0,838165
02	3	0,839742
03	3	0,807159
04	3	0,835978
05	3	0,851022
01	4	0,849396
02	4	0,879417
03	4	0,792706
04	4	0,860887
05	4	0,853139
01	5	0,845676
02	5	0,845342
03	5	0,793865
04	5	0,865843
05	5	0,879200
01	6	0,843900
02	6	0,806214
03	6	0,829652
04	6	0,822622
05	6	0,868920
01	7	0,843290
02	7	0,715067
03	7	0,814755
04	7	0,842029
05	7	0,855083
01	8	0,849012
02	8	0,858046
03	8	0,793855
04	8	0,869690
05	8	0,896945
01	9	0,846329
02	9	0,859715
03	9	0,586426
04	9	0,892969
05	9	0,821227
01	10	0,858038
02	10	0,868104
03	10	0,027608
04	10	0,027608
05	10	0,027608

9.4 RESULTADOS: CONJUNTO DE DADOS DE BIGRAMAS

TAB. 9.17: Resultado dos Experimentos com vetores de bigramas para redes classificadoras com 1 camada oculta

Experimento	**Macro F1**
F0_1	0,824217
F0_2	0,825132
F0_3	0,826143
F0_4	0,824999
F0_5	0,825361
F0_6	0,825591
F0_7	0,825554
F0_8	0,824577
F0_9	0,825494
F1_0	0,823325
F1_1	0,823662
F1_2	0,825482
F1_3	0,825373
F1_4	0,826421
F1_5	0,825185
F1_6	0,825798
F1_7	0,825232
F1_8	0,825492
F1_9	0,825947
F2_0	0,825257

TAB. 9.18: Resultado geral das redes classificadoras até 10 camadas ocultas, utilizando o conjunto de dados de vetores de bigramas

Experimento	**Qtd. Camadas Ocultas (Classificadora)**	**Macro$_{F_1}$**
01	2	0,899832
02	2	0,909714
03	2	0,924747
04	2	0,923058
05	2	0,929258
01	3	0,912121
02	3	0,904694
03	3	0,900467
04	3	0,905617
05	3	0,894867
01	4	0,900463
02	4	0,861658
03	4	0,906649
04	4	0,931399
05	4	0,888904

10 ANEXOS