UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DEPARTAMENTO ACADÊMICO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

LEONARDO CORREIA SATO

DEEP LEARNING NA SEGURANÇA COMPUTACIONAL: DETECÇÃO INTELIGENTE DE CÓDIGOS MALICIOSOS

TRABALHO DE CONCLUSÃO DE CURSO

PATO BRANCO
2018

LEONARDO CORREIA SATO

DEEP LEARNING NA SEGURANÇA COMPUTACIONAL: DETECÇÃO INTELIGENTE DE CÓDIGOS MALICIOSOS

Trabalho de Conclusão de Curso 2, apresentado à disciplina de Trabalho de Conclusão de Curso 2, do Curso de Engenharia de Computação da Universidade Tecnológica Federal do Paraná - UTFPR, Câmpus Pato Branco, como requisito parcial para obtenção do título de Engenheiro da Computação.

Orientador: Prof. Dr. Dalcimar Casanova

Coorientador: Prof. Dr. Bruno César Ribas

PATO BRANCO

2018

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Câmpus Pato Branco
Departamento Acadêmico de Informática
Curso de Engenharia de Computação

UTFPR
UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

TERMO DE APROVAÇÃO

Às 13 horas e 50 minutos do dia 12 de dezembro de 2018, na sala V108, da Universidade Tecnológica Federal do Paraná, Câmpus Pato Branco, reuniu-se a banca examinadora composta pelos professores Dalcimar Casanova (orientador), Bruno Cesar Ribas (coorientador), Fábio Favarim e Kathya Silvia Collazos Linares para avaliar o trabalho de conclusão de curso com o título **Deep Learning na segurança computacional: detecção inteligente de códigos maliciosos**, do aluno **Leonardo Correia Sato** matrícula 01260790, do curso de Engenharia de Computação. Após a apresentação o candidato foi arguido pela banca examinadora. Em seguida foi realizada a deliberação pela banca examinadora que considerou o trabalho aprovado.

_____	_____
Prof. Dalcimar Casanova Orientador (UTFPR)	Prof. Bruno Cesar Ribas Coorientador (UTFPR)
_____	_____
Prof. Fábio Favarim (UTFPR)	Profa. Kathya Silvia Collazos Linares (UTFPR)
_____	_____
Profa. Beatriz Terezinha Borsoi Coordenador de TCC	Prof. Pablo Gauterio Cavalcanti Coordenador do Curso de Engenharia de Computação

A Folha de Aprovação assinada encontra-se na Coordenação do Curso.

RESUMO

SATO, Leonardo Correia. *Deep Learning* na segurança computacional: detecção inteligente de códigos maliciosos. 2017. 69f. Trabalho de Conclusão de Curso de Bacharelado - Curso de Engenharia de Computação, Universidade Tecnológica Federal do Paraná, Câmpus Pato Branco. Pato Branco, 2017.

O aumento na quantidade de malwares e suas famílias amplificou os problemas de detecção automática e classificação de suas novas variantes. Na medida que as ameaças computacionais evoluem, também cresce a necessidade de mecanismos de defesa efetivos para proteger os dispositivos. Porém, torna-se progressivamente mais difícil blindar terminais de serem infectados. São necessárias ferramentas que identifiquem os códigos maliciosos residentes nos sistemas para lidar com os casos nos quais esquemas de prevenção contra malwares não funcionarem. Neste trabalho de conclusão de curso, é investigado em etapas a aplicação de uma arquitetura de Redes Neurais Profundas (do inglês *Deep Neural Network*, DNN) para detecção de malwares com base em suas chamadas de funções do sistema operacional. A estrutura do modelo *Deep Learning* utiliza um *AutoEncoder* e as sequências de chamadas dos malwares para extração de características, formando vetores que funcionam como as assinaturas dos códigos maliciosos. Amostras de códigos maliciosos e benignos foram utilizadas para treinar e testar os classificadores. A efetividade do *AutoEncoder* construído em facilitar a correta classificação dos códigos maliciosos ficou evidente pelos resultados obtidos com os classificadores.

Palavras-chave: Redes Neurais, Malwares, Deep Learning, TensorFlow.

ABSTRACT

SATO, Leonardo Correia. Deep Learning in Computer Security: Intelligent Detection of Malicious Code. 2017. 69f. Work of Course Completion - Computer Engineering Major, Federal Technological University of Paraná, Pato Branco Campus. Pato Branco, 2017.

The increase in the amount of malware and their families amplified the problems of automatic detection and classification of their new variants. As computer security threats evolve, so does the need for effective defense mechanisms to protect the devices. However, it becomes progressively more difficult to protect terminals from being infected. Thus, tools which identify resident malicious codes are required for handling post-infection systems. In this work of course completion, the application of a Deep Neural Network (DNN) architecture to detect malwares based on its operational system processes is investigated. The Deep Learning framework proposed implements a AutoEncoder and utilizes API call sequences to extract features, forming vectors that function as signatures of malicious codes. Samples of malicious and benign codes were obtained to train and test the classifiers. The effectiveness of AutoEncoder built to facilitate the correct classification of the malicious codes was made evident by the results obtained from the classifiers.

Keywords: Neural Networks, Malwares, Deep Learning, TensorFlow.

LISTA DE FIGURAS

LISTA DE ACRÔNIMOS E SIGLAS

AE	*AutoEnconder*.
API	*Application Programming Interface*.
DGA	*Domain Generation Algorithm*.
DNN	*Deep Neural Network*.
IA	Inteligência Artificial.
IoT	*Internet of Things*.
KNN	*k-nearest neighbors*.
Malware	*Malicious Software*.
PE	*Portable Executable*.
RNA	Redes Neurais Artificiais.
SVM	*Support Vector Machine*.
VLSI	*Very-Large-Scale-Integration*.
Win-API	API do Windows.

SUMÁRIO

1 INTRODUÇÃO

Ameças de segurança para ambientes virtuais não são um tema novo, tornando-se proeminentes nos anos recentes. Programas criminosos representam uma grave ameaça para sistemas computacionais quando se disseminam pois comprometem a segurança, integridade e funcionalidade dos sistemas. Os prejuízos financeiros causados por esses softwares maliciosos podem alcançar o valor de bilhões de dólares, especialmente no que se refere aos que não são detectados (HARDY *et al.*, 2016).

Um malware (abreviação de "*malicious* software", que significa software malicioso em inglês) é um programa feito para danificar ou destruir funcionalidades de um sistema operacional(LI *et al.*, 2015). O malware compromete seu funcionamento através da adição, modificação ou remoção de alguma parte do código no sistema para cumprir propósitos nocivos. O comportamento esperado do computador é então alterado, fazendo-o realizar atividades prejudiciais aos usuários autorizados (LI *et al.*, 2015). Um exemplo de uma vítima notória dos malwares é o Adobe Flash, que foi alvo frequente de ataques cibernéticos devido a sua popularidade e vulnerabilidades (JUNG *et al.*, 2015). Em razão das questões financeiras envolvidas, detecção de malwares é uma área de importância para a indústria.

Diversos pesquisadores desenvolveram softwares contra malwares como resposta a demanda por segurança em ambientes virtuais. A maioria utiliza o método baseado em assinaturas simples para detecção (HARDY *et al.*, 2016), mas pode ser facilmente ultrapassado pelos criminosos que produzem malware utilizando técnicas diversas. Além disso, arquivos infecciosos estão sendo espalhados em uma taxa de milhares por dia, o que torna esse método ainda menos efetivo.

Como consequência, métodos inteligentes de detecção de malware começaram a ser investigados pela possibilidade de serem mais eficientes. Porém, eles foram baseados em técnicas rudimentares da aprendizagem de máquina que se provaram insuficientes para cenários envolvendo problemas mais complexos (HARDY *et al.*, 2016). São necessários métodos mais aptos em lidar com o aumento da complexidade dos problemas envolvendo detecção inteligente de malwares em larga escala. Neste sentido, entra no cenário atual a técnica conhecida como Redes Neurais Profundas (do inglês *Deep Neural Network*, ou simplesmente referenciada como *Deep Learning*).

O conceito de *Deep Learning* trata da utilização de múltiplas camadas escondidas em redes neurais que são previamente treinadas com uma grande quantidade de dados para conseguirem detectar propriedades diversas e criarem um modelo de

classificação adequado. Somente alcançou popularidade recentemente com relação aos outros métodos, sendo que antes o uso de *Deep Learning* era mais restrito aos que possuíssem habilidade em programação, conhecimentos matemáticos e acesso a dispositivos com alto poder de processamento. Os métodos tornaram-se populares com ajuda dos avanços tecnológicos recentes que disseminaram a utilização através da facilitação do uso. A capacidade do *Deep Learning* de generalizar soluções trouxe diversos avanços atuais na área da inteligência artificial (IA), especificamente para o aprendizado de máquina. Os sucessos obtidos com estas técnicas sugerem que o *Deep Learning* poderia ser usado efetivamente para a detecção de malwares, uma motivação para este trabalho.

1.1 OBJETIVOS

1.1.1 OBJETIVO GERAL

Desenvolver um método utilizando *Deep Learning* que otimize a detecção de códigos maliciosos, auxiliando as ferramentas protetoras de sistemas virtuais contra ameaças externas.

1.1.2 OBJETIVOS ESPECÍFICOS

- Criar e disponibilizar a base de dados dos arquivos benignos analisados, fornecendo uma base pronta para qualquer pessoa que deseje seguir a mesma abordagem na extração de características.
- Disponibilizar os códigos desenvolvidos de forma livre, incentivando a utilização de *Autoencoders*(um tipo especial de rede neural) e contribuindo com a detecção de malwares.
- Desenvolver o protótipo de um sistema detector de malwares que obtenha uma nota acima de 60% para a métrica que será considerada, com ou sem o uso do *Autoencoder*.

1.2 JUSTIFICATIVA

Houve um aumento nas ameaças virtuais entre 2015 e 2016. Crimes cibernéticos ocorreram contra mais vítimas de destaque e geraram lucros altos para os criminosos (WUEEST, 2017). Em 2016, ocorreram diversos ataques virtuais contra sistemas financeiros do mundo inteiro. Esse evento foi histórico em razão dos fortes indícios de que os ataques teriam recebido apoio do governo de um país (WUEEST, 2017). A Figura 1 ilustra o aumento dos incidentes criminosos envolvendo malwares entre os períodos de 2011 e 2017 (RANA *et al.*, 2017).

Figura 1 – Quantidades em milhões de incidentes envolvendo malwares, entre 2011 e 2017
Fonte: Adaptado de Bloomberg Intelligence (2017).

Foi observado um aumento de redirecionamento para sites falsos até roubo de credenciais de contas bancárias em aparelhos móveis, com malwares que realizam essa função afetando ao menos 170 aplicativos (WUEEST, 2017). Estimativas apontam que durante esse período os ataques cibernéticos podem ter levado a prejuízos em até 300 milhões de dólares (WUEEST, 2017). A Figura 2 mostra um gráfico com dados conhecidos de prejuízos orçamentários em razão de malwares para organizações no setor empresarial de vários países(PONEMON INSTITUTE, 2016).

A popularidade das tecnologias as colocam na mira de criminosos com a intenção de atacá-las com softwares maliciosos, trazendo grandes prejuízos. O tempo é sempre escasso na luta sem fim contra ameaças virtuais, pois os seus desenvolvedores mal-intencionados sempre pesquisam novas técnicas de ataque e aperfeiçoam as que já utilizam. É preciso manter a segurança computacional atualizada e desenvolver novas técnicas para conseguir superar os avanços feitos pelos criminosos virtuais e manter a segurança dos sistemas informáticos.

Métodos detectores de malware baseados em assinaturas ou comportamento estão obsoletos e não são capazes de enfrentar ameaças virtuais da nova geração

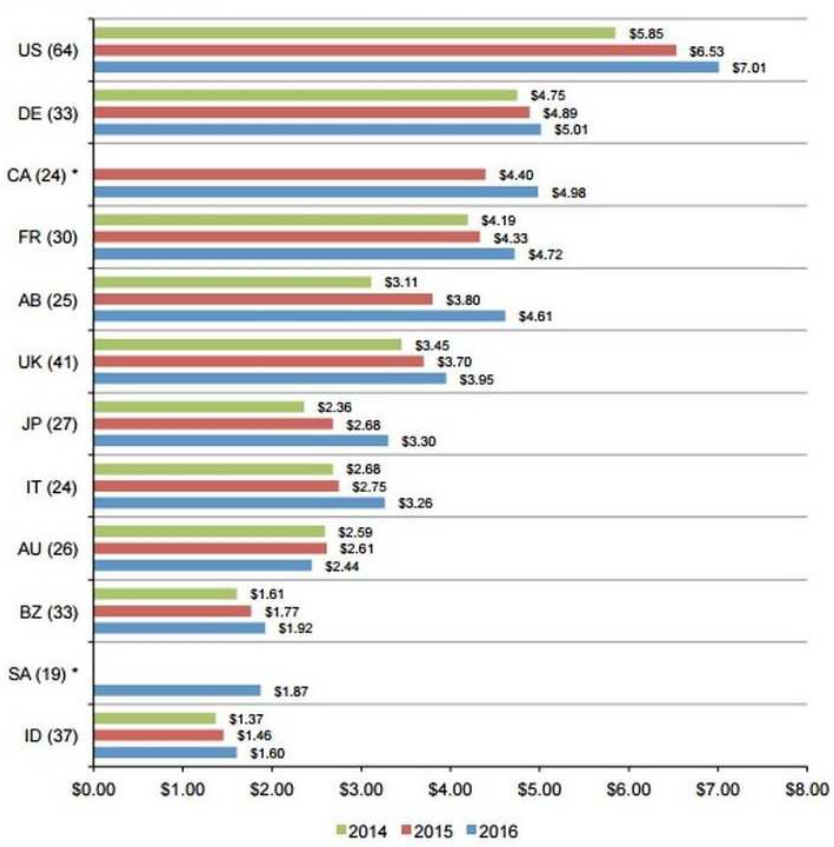

Figura 2 – A média total (em milhões) do custo de um vazamento de dados para organizações, em três anos

Fonte: Ponemon Institute (2016)

que utiliza hashes mutantes, mecanismos de ofuscação, autopropagação e componentes inteligentes na sua ação maliciosa (SCOTT, 2017). Aprendizado de máquina, em contraste, é uma tecnologia atual da área de Inteligência artificial que é capaz de oferecer uma característica preditiva para a segurança virtual das organizações, permitindo que obtenham uma vantagem necessária sobre os seus inimigos virtuais através da detecção de códigos maliciosos antes que eles atuem sobre os sistemas (SCOTT, 2017).

Deep Learning é uma técnica do aprendizado de máquina ligada à redes neurais que vem sendo utilizada para várias aplicações, desde processamento de imagem até a vencer campeões de jogos de tabuleiro. É possível aplicar esse método para um problema que envolva reconhecimento de características específicas para classificação, ou seja, uma solução para detecção de códigos maliciosos pode ser implementada.

Pesquisas de redes neurais artificiais tentam imitar a rede neural de um cérebro e a biologia humana desde 1950, quando foram inventados e aplicados os primeiros algoritmos do aprendizado de máquina (SCOTT, 2017). Tentativas de aplicar algo-

ritmos de inteligência artificial para a segurança cibernética começaram entre 2000 e 2010, porém ainda dependiam de assinaturas dos códigos maliciosos (SCOTT, 2017). Detecção de malware baseada em assinaturas não é escalável em um cenário de milhões de novas assinaturas todos os dias, o que forçou as instituições que lidam com a segurança computacional a buscar novos métodos que não usam assinaturas para identificar malwares (SCOTT, 2017). Um dos novos métodos que está sendo estudado por pesquisadores é o *Deep Learning*, cujo desempenho por quantidade de dados costuma ser maior que o encontrado em métodos mais antigos. A Figura 3 ilustra um gráfico simples comparando o desempenho de métodos antigos contra o desempenho dos métodos de *Deep Learning*, em razão da quantidade de dados disponíveis.

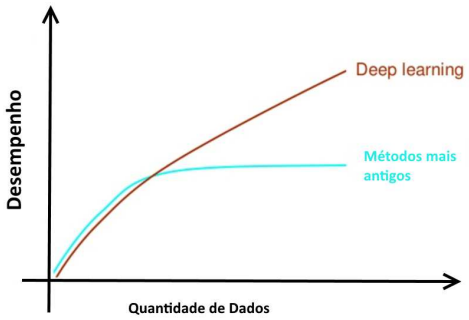

Figura 3 – Desempenho do *Deep Learning* comparado com métodos tradicionais do aprendizado de máquina

Fonte: Autoria própria.

Deep Learning utiliza uma arquitetura de redes neurais com múltiplas camadas para adquirir uma capacidade maior de identificar padrões(HARDY *et al.*, 2016), realizando uma mineração extensiva de dados. Essas arquiteturas superam a dificuldade de reconhecimento de característica utilizando um pré-tratamento nas suas múltiplas camadas de detecção de características, do nível mais baixo até o mais alto para construir um modelo preciso de classificação (HARDY *et al.*, 2016). Essa propriedade é a que torna interessante para pesquisadores, que aplicam *Deep Learning* em várias áreas: processamento de linguagem natural, visão computacional, e recentemente em pesquisas para reconhecimento de códigos maliciosos (HARDY *et al.*, 2016).

Normalmente uma grande quantidade de amostras de códigos maliciosos será preciso para treinar o sistema utilizando *Deep Learning*. Vai ser construído um conjunto de amostras de códigos misturando malwares com códigos não-maliciosos para que sejam tratados e corretamente identificados pelo sistema que será projetado, ensinando-o a comportar-se como identificador de malwares.

Não é difícil encontrar os códigos maliciosos necessários para construir o conjunto de amostras através da Internet. É possível requisita-los de instituições que

trabalham com programas de antivírus e segurança de ambientes virtuais em geral. Exemplos são: Comodo(COMODO GROUP, 2018) ou Symantec(SYMANTEC CORPORATION, 2018). Além disso, existem comunidades abertas para pesquisadores que dão livre acesso à base de dados com várias amostras de malwares, como a VirusShare(FORENSICATION, 2018) ou MalShare(CUTLER, 2018).

Também não é necessário desenvolver do zero uma rede neural do tipo *Deep Learning*, mesmo sendo um método que recentemente adquiriu popularidade. Existem APIs para ajudar na construção de aplicações com *Deep Learning*. Alguns exemplos de APIs atualmente disponíveis são: DeepDetect(JOLIBRAIN,) da JoliBrain, Watson APIs(IBM, 2018) da IBM, Caffe(BERKELEY AI RESEARCH, 2014) da Berkeley AI Research e o TFLearn(DAMIEN, 2016) que pertence a uma biblioteca de softwares de código aberto chamada TensorFlow. O escolhido para este trabalho foi o TensorFlow, devido a sua maior popularidade e documentação em comparação aos demais.

1.3 ESTRUTURA DO TRABALHO

No próximo capítulo será apresentado o referencial teórico. A pesquisa construída é focada em conhecimentos atuais dos seguintes tópicos: Inteligência Artificial, Aprendizado de Máquina, Redes Neurais, *Deep Learning*, Segurança Computacional, Malwares, *API Calls*, Extração de Características e TensorFlow. Trabalhos relacionados são apresentados com exposição de avanços recentes na área de aplicação de *Deep Learning* para detecção de malwares. O modelo deste trabalho é apresentado no capítulo seguinte, descrevendo sua implementação que consiste das etapas ordenadas: aquisição de base de amostras contendo códigos maliciosos e não-maliciosos, elaboração e construção de um sistema *Deep Learning* para detectar malwares utilizando autoencoder, e pré-processamento das entradas no sistema para extração de características. Os últimos capítulos são os resultados obtidos através da comparação com outras técnicas exploradas em segurança computacional e o resultado final deles, com as conclusões e possíveis desenvolvimentos futuros que podem ser explorados.

2 REFERENCIAL TEÓRICO

Nesta seção são organizados todos os conceitos que são precisos para o planejamento deste trabalho e seu desenvolvimento. Estão relacionados com as atividades necessárias para este trabalho, desde a etapa de construção do modelo proposto até a obtenção dos resultados. Os conceitos abordados serão: Explicações sobre Inteligência Artificial, a Teoria envolvendo Aprendizado de Máquinas, noções de Redes Neurais para *Deep Learning*, Segurança Computacional, tipos diferentes de Malwares, e extração de características dos Malwares. Os metadados extraídos serão as sequências de funções chamadas pela API do Windows. O TensorFlow é utilizado para a composição do sistema *Deep Learning* proposto, que vai otimizar classificadores existentes com um pré-processamento dos metadados extraídos.

2.1 INTELIGÊNCIA ARTIFICIAL

A Inteligência Artificial é a área que engloba uma alta gama de conhecimentos cujo propósito é a criação de programas computacionais com a habilidade de aprenderem, raciocinarem e agirem de forma que obtenham a maior chance possível de sucesso para os objetivos pelos quais foram projetados. Vem sendo utilizada para transformar como as pessoas acessam e utilizam suas informações na rede. Tornou-se estrategicamente importante para a defesa de nações e a segurança de infraestruturas críticas de finanças, energia e comunicações (CYLANCE, 2017).

Pode-se dividir a área de Inteligência Artificial em três partes: Superinteligência, Geral e Fraca (CYLANCE, 2017).

- A Superinteligência foca no desenvolvimento de máquinas superiores aos seres humanos em virtualmente todos os sentidos e pode ser descrita como especulativa no presente momento.

- A Inteligência Artificial Geral, também conhecida como forte, é uma máquina que é tão inteligente e capaz quanto um ser humano, ou seja, uma máquina que seria capaz de passar o famoso teste de Turing.

- A Fraca refere-se ao uso de um computador para processar grandes quantidades de dados e detectar padrões que seriam difíceis ou inviáveis para um ser humano analisar.

Inteligência Artificial Fraca é capaz de superar pessoas apenas em tarefas

específicas, como jogos ou detectar anomalias em um sistema, e essa é a abordagem que será o foco deste trabalho. Os avanços obtidos nessa área foram vistos nas áreas de utilidades eletrônicas, manufatura, saúde, educação e outras que trouxeram evidências sugerindo aumento de desempenho e qualidade em sistemas nos quais a Inteligência Artificial Fraca foi implementada (BUGHIN *et al.*, 2017).

Atualmente, máquinas integradas com Inteligência Artificial cumprem diversas tarefas como reconhecimento de padrões complexos, sintetização de informações, formação de conclusões e previsões. A Inteligência Artificial Fraca continua observando progresso significativo, sendo expandida para um número crescente de áreas mesmo diante de problemas técnicos como viés devido a conjuntos específicos de dados (BUGHIN *et al.*, 2017).

A maioria dos avanços na abordagem de Inteligência Artificial Fraca são de uma subdisciplina chamada de Aprendizado de Máquina, cuja meta é criar máquinas capazes de aprender com a aplicação de algoritmos sobre dados. Aprendizado de Máquina foca e somente lida com problemas que são solucionáveis através da obtenção de dados relevantes que possam existir e serem adquiríveis, ou seja, problemas como os de segurança computacional (CYLANCE, 2017).

2.2 APRENDIZADO DE MÁQUINA

O Aprendizado de Máquina, chamado de *Machine Learning* em inglês, é a área de estudo dos métodos para que programas computacionais obtenham a capacidade de aprender a resolver tarefas de diversos tipos: desde aquelas consideradas difíceis para serem solucionadas de forma satisfatória por especialistas humanos, custosas para códigos mais simples de programação ou até mesmo as atividades mundanas para o ser humano mas cuja solução via máquina seria preferível(BHATTACHARYYA; KALITA, 2014).

Formalmente, aprendizado de máquina é definido como o complexo processo de utilizar recursos computacionais na implementação de algoritmos de aprendizado para o reconhecimento automático de padrões e a tomada de decisões consideradas inteligentes através de amostras de dados treinadores. Aprendizado é o processo de construir um modelo científico depois de adquirir conhecimento de uma amostra ou conjunto de dados (DUA; DU, 2011). Trata das mesmas questões pesquisadas nos campos da estatística, mineração de dados e psicologia. Porém, sua ênfase é diferente: está primariamente interessada na precisão e efetividade dos sistemas de

aprendizado (BHATTACHARYYA; KALITA, 2014).

Em resumo, aprendizado de máquina utiliza métodos avaliativos para eliminar a necessidade de contínuo controle externo humano e permitir ao programa aprimorar seu desempenho através da progressiva análise de um problema que é solucionado em repetidas instâncias. As tarefas abordadas podem ser divididas nas seguintes categorias (BHATTACHARYYA; KALITA, 2014):

- Tarefas nas quais não há especialistas humanos, como sistemas que possuem requerimentos que são inviáveis para um ser humano prever devido a quantidade de dados que precisam ser analisados.

- Tarefas em que há especialistas humanos, mas eles não sabem como explicar de forma satisfatória todos os passos que eles tomam para resolve-las. Por exemplo, entendimento de linguagem natural.

- Tarefas nas quais o contexto muda rapidamente, como as que envolvem a bolsa de valores, compras de consumidores ou trocas de câmbio.

- Tarefas que exigem customização diferenciada para cada usuário, como filtros de mensagens eletrônicas.

Adicionalmente, tarefas de aprendizado podem ser classificadas com relação a distinção entre as empíricas e analíticas. Aprendizado empírico é correspondente ao supervisionado por necessitar de experiência externa, enquanto aprendizado analítico não precisa e portanto corresponde ao não-supervisionado. Para o aprendizado supervisionado inicialmente utiliza-se exemplos pré-classificados por um agente externo para a formação de regras que vão auxiliar na classificação de futuras amostras dos dados, enquanto que no aprendizado não-supervisionado o sistema aprende apenas o comportamento dos dados e realiza o agrupamento deles através de regras baseadas nesse comportamento, não necessitando da rotulação de amostras que é feita pela intervenção humana (BHATTACHARYYA; KALITA, 2014).

As duas formas de aprendizado trabalham de acordo com as seguintes premissas: instâncias normais de dados são mais frequentes que anomalias e são estatisticamente diferentes. Essas suposições precisam ser verdadeiras, do contrário haverá uma alta taxa de alarmes falsos nos métodos de aprendizados (BHATTACHARYYA; KALITA, 2014).

Métodos de aprendizado utilizam padrões de treino para decidir ou estimar a forma de modelos classificadores, que podem ser paramétricos ou não-paramétricos,

com o propósito de reduzir os erros de classificação durante o treinamento. Diferenciam-se uns dos outros pela seleção do paradigma de aprendizado, pelos seus parâmetros e pela expressão do seu erro de aprendizado. O erro de aprendizado é o resultado da comparação entre os resultados do paradigma de aprendizado com informações conhecidamente verdadeiras (DUA; DU, 2011).

Alguns métodos comuns para aprendizado supervisionado são: Árvores de Decisão, Redes Neurais Artificiais (RNA) e Máquina de vetores de suporte. Já métodos famosos de aprendizado não-supervisionado são: Agrupamento (do inglês *Clustering*), mineração de padrões frequentes ou de valores aberrantes (BHATTACHARYYA; KALITA, 2014). A Figura 4 compara as duas formas de aprendizado, ilustrando elementos que elas possuem em comum nas suas estruturas ou que são exclusivos de cada aprendizado.

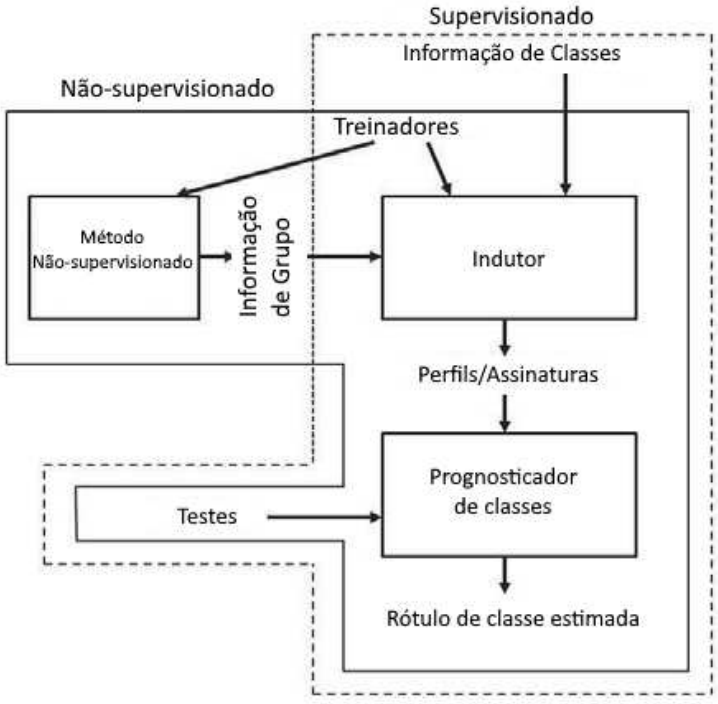

Figura 4 – Métodos supervisionados vs não-supervisionados
Fonte: Adaptado de Bhattacharyya (2014, p. 60).

Além disso, há métodos híbridos que são considerados semi-supervisionados como: modelos generativos, abordagens heurísticas e outros (DAUMÉ III, 2017). Métodos semi-supervisionados são aplicados em problemas nos quais há um conjunto de dados pré-classificados, mas de tamanho reduzido. Estes métodos híbridos possuem diversas aplicações em detecção de anomalias cibernéticas (DUA; DU, 2011).

Em geral, qualquer método em todas as formas de aprendizado de máquina

terá que contemplar os melhores procedimentos para conceber uma implementação concreta de um problema abstrato de aprendizado. Haverá um grande impacto no desempenho do sistema de aprendizado dependendo de como o problema abstrato for transformado em uma tarefa concreta implementada. As seguintes questões de requisitos devem ser levantadas sobre o sistema de aprendizado (DAUMÉ III, 2017):

(i) *Onde pode ocorrer erro no sistema?* As etapas básicas na construção de um sistema de aprendizado de máquina são: coletar dados, escolher características, uma família de modelo, os dados de treino, treinar o modelo criado e avalia-lo segundo os resultados de testes. Em cada uma dessas etapas, algo pode dar errado. É preciso prever em qual etapa o erro ocorre para isolar sua causa, neste sentido são feitas as próximas perguntas como estratégias para encontrar a causa.

(ii) *O algoritmo de aprendizado está implementado corretamente?* O algoritmo pode não estar otimizando o sistema da forma como foi planejado. Para testar essa hipótese é preciso utilizar um conjunto de dados no qual o comportamento desejado pelo sistema de aprendizado já seja conhecido. Pode-se comparar com uma implementação de referência para verificar se otimização esperada pelo algoritmo é aquela que ocorre na realidade.

(iii) *Sua representação do problema é adequada?* É importante eliminar aspectos que sejam irrelevantes ou redundantes para as tarefas, adicionando e mantendo apenas os bons aspectos para tornar o sistema consistente e mais eficiente no aprendizado. Atributos desnecessários afetam o erro do treinamento, assim também como a falta de atributos que sejam importantes.

(iv) *Há informação suficiente para o sistema?* Treinar com apenas um pouco a menos dos dados usualmente utilizados pode revelar diferenças importantes na performance. Se a performance diminuir bastante por causa da falta desses poucos dados, então é provável que o sistema precise de mais dados. Se a performance sofrer de forma insignificante, então pode ser que haja uma saturação de informação.

O sistema de aprendizado desenvolvido é implementado com exito quando as questões acima são respondidas de forma satisfatória. Essas questão serão respondidas no sistema de aprendizado de máquina que será apresentado neste trabalho: um modelo semi-supervisionado da família que utiliza múltiplas redes neurais empilhadas chamada de *Deep Learning*.

2.3 REDES NEURAIS

Antes de entrar no tópico de *Deep Learning*, primeiro é preciso explicar o que são Redes Neurais. As Redes Neurais são processadores distribuídos em larga escala e paralelamente uns com os outros, sendo cada processador uma unidade simples que possui tanto a capacidade de armazenar conhecimentos obtidos via experiência quanto a habilidade de tornar as informações que forem armazenadas disponíveis para utilização (HAYKIN, 2008).

Possui dois atributos similares aos do cérebro: a informação é obtida em uma rede através de um processo de aprendizado em conjunto de interação com o contexto experimental, e a forma de armazenamento de informações adquiridas é feita via as forças de conexão entre neurônios chamadas de pesos sinápticos (HAYKIN, 2008).

Um neurônio é uma unidade processadora de informações que é fundamental para a operação de uma rede neural. Existem três elementos básicos em um neurônio artificial (HAYKIN, 2009):

- Conexões sinápticas caracterizadas por um peso próprio. Os valores de peso podem ser tanto negativos quanto positivos. Um sinal de entrada x_j na conexão j do neurônio k é multiplicado pelo peso sináptico w_{jk}.

- Uma junção aditiva que soma os sinais de entrada, pesando-os pelas conexões sinápticas do neurônio através de combinações lineares.

- Uma função de ativação que vai limitar a amplitude de saída do neurônio, reduzindo o seu alcance até um valor finito. Geralmente o alcance da saída de um neurônio fica entre o intervalo de [0,1] ou [-1,1].

A Figura 5 mostra um exemplo de um neurônio. O modelo da Figura 5 também inclui um bias aplicado externamente, chamado de b_k, que é um valor associado em cada entrada dos nós da rede neural. Seu propósito é aumentar ou diminuir a entrada total da função de ativação dependendo do seu sinal ser positivo ou negativo, ampliando a capacidade da rede neural de encontrar a função que retorna as soluções desejadas. Descrevendo matematicamente:

$$\mu_k = \sum_{j=1}^{m} w_{kj} x_j \tag{1}$$

$$y_k = \varphi(\mu_k + b_k) \tag{2}$$

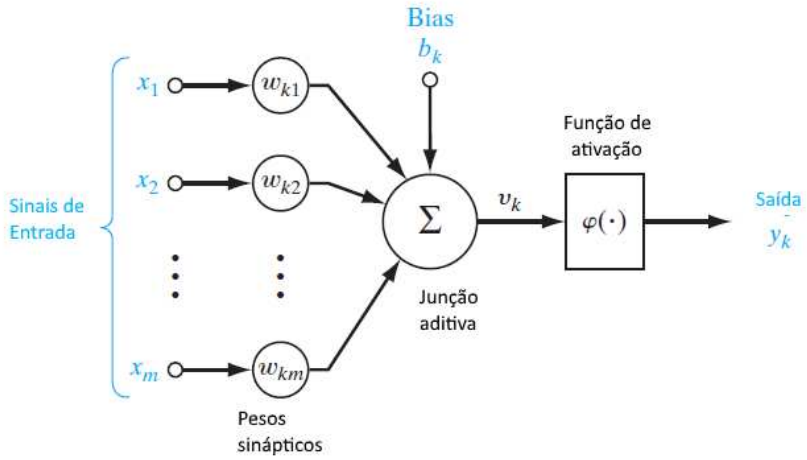

Figura 5 – Exemplo de um neurônio não-linear para uma Rede Neural
Fonte: Traduzido de Haykin (2009, p. 11).

As equações 1 e 2 possuem as mesmas variáveis vistas na Figura 5 e representam suas relações matemáticas. Além do modelo de neurônio, é preciso definir o método de aprendizado. Esse algoritmo pode ser tanto supervisionado, não-supervisionado ou semi-supervisionado. São os paradigmas de aprendizado vistos na seção anterior.

O algoritmo de aprendizagem nas redes neurais é aquele que vai modificar os seus pesos sinápticos de forma a atingir o propósito do sistema projetado. Outra possibilidade no método é a modificação de sua topologia através da criação de novas conexões entre os neurônios. A função do algoritmo é generalizar, isto é, criar saídas corretas para entradas que não foram apresentadas ao sistema de aprendizado durante o seu treinamento. Redes Neurais conseguem solucionar tarefas maciças devido ao seu poder de generalização e seu arranjo paralelamente distribuído em abundância (HAYKIN, 2008).

As Redes Neurais aprendem através de determinados métodos, que podem ter os seus tipos mais básicos descritos resumidamente como (HAYKIN, 2009):

- Associação de padrões: aprendizado supervisionado (heteroassociação) ou não-supervisionado (autoassociação). Autoassociação envolve ensinar um conjunto de padrões ao sistema para permitir que ele recupere os padrões de versões distorcidas dos mesmos. Heteroassociação, em contraste, busca permitir que o sistema recupere um padrão de outro padrão diferente.

- Reconhecimento de padrões: Processo no qual um sinal ou padrão é nomeado para um dos números prescritos de classes. Redes Neurais que utilizam esse tipo de método podem existir em uma das duas formas: utilizando

uma parte não-supervisionada da rede para extração de características e mais uma parte supervisionada para classificação, ou uma rede *feedforward*(pré-alimentada) utilizando um algoritmo supervisionado no qual a extração de características é feita por uma ou mais camadas escondidas na Rede Neural.

- Aproximação de funções: Se existe uma função desconhecida para resolver um problema, mas há instâncias solucionadas do mesmo então é possível criar uma outra função cujo o mapeamento de entrada pela saída possa ser feito pela Rede Neural de modo que seja perto o suficiente da função desconhecida para que o erro de aproximação seja um valor pequeno o suficiente para considerar a função implementada como candidata para aprendizado supervisionado.

- Controle de processos: Pode-se transformar a Rede Neural em um controlador de sistema utilizando *feedback*(pós-alimentada), para manter o mesmo em uma situação controlada. Esse método de aprendizado toma vantagem da distribuição paralela das redes neurais para realizar o controle de vários atuadores ao mesmo tempo, com a rede neural tratando da não-linearidade e os ruídos do sistema na medida que também consegue otimizar planejadamente o sistema por tempo prolongado.

- Filtragem espacial: É o método de aprendizado que distingue propriedades espaciais do sinal-alvo de ruído causado pelo ambiente. Compatível com Redes Neurais cujo propósito é o mapeamento de características para sistemas auditórios, por exemplo.

Além dos métodos de aprendizado citados, é preciso entender as arquiteturas da Rede Neural. No geral, existem três tipos fundamentalmente diferentes de classes de arquiteturas de rede: Redes Pré-Alimentadas de Uma Camada como na Figura 6, Redes Pré-Alimentadas de Múltipla Camadas e Redes Recorrentes (HAYKIN, 2009).

Nas Redes do primeiro e segundo tipo, os neurônios são organizados em forma de camadas. Na primeira forma, tempos apenas uma camada de entrada e uma de saída com os neurônios. A segunda forma apresenta uma ou mais camadas escondidas, ou seja, camadas que não são vistas nem pela entrada ou saída. As camadas escondidas tem a função de intervir entre a entrada e saída de alguma forma que seja considerada útil, geralmente extraindo estatísticas de maior ordem da entrada devido ao aumento da dimensão das interações neurais. As dimensões aumentam na medida que há mais camadas extras escondidas.

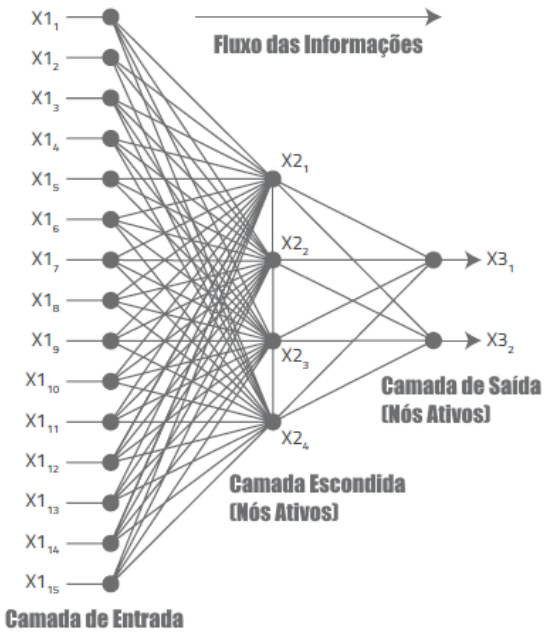

Figura 6 – **Arquitetura Genérica de uma Rede Neural**
Fonte: Traduzido de Cylance (2017, p. 118).

A Figura 7 ilustra alguns exemplos de redes neurais que pertencem ao primeiro e segundo tipo (VEEN, 2016). O *Perceptron* pertence ao primeiro tipo e as outras três percentem ao segundo tipo. Por fim, as Redes Recorrentes são diferentes das Pré-Alimentadas porque possuem pelo menos um laço de *feedback* nos seus neurônios, sendo que essas redes são pós-alimentadas independentemente da quantidade de camadas escondidas que possuírem (HAYKIN, 2009).

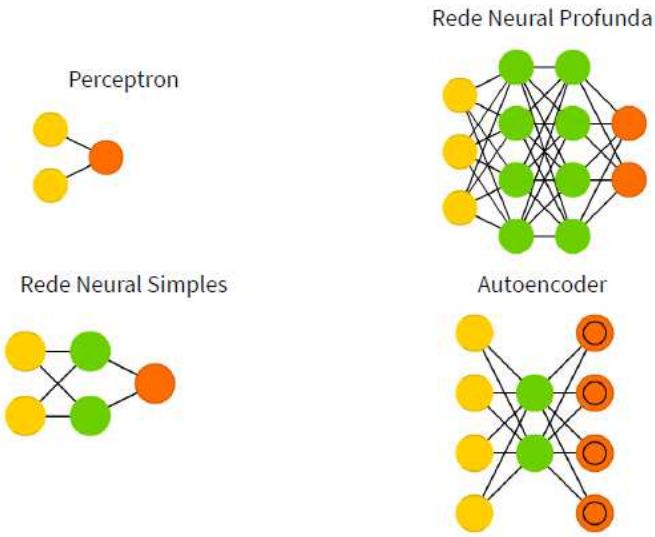

Figura 7 – **Arquiteturas de diferentes Redes Neurais**
Fonte: Traduzido de Veen (2016).

A arquitetura utilizada neste trabalho é uma Rede Pré-alimentada de Múltiplas Camadas utilizando autoassociadores. O motivo pelo qual as redes neurais foram escolhidas para a implementação deste trabalho vai desde de fatores como o Estado da Arte, que será visto na seção de trabalhos relacionados, até as vantagens que são obtidas com a utilização das Redes Neurais. A implementação de redes neurais como modelo do sistema de aprendizado permite a aquisição dos seguintes atributos e funcionalidades (HAYKIN, 2008):

(i) Não-linearidade: Os neurônios artificiais da rede tem a opção de serem lineares ou não. Uma Rede Neural com neurônios não-lineares é considerada não-linear também. No caso de Redes Neurais, a não-linearidade é distribuída pela rede e é essencial se a geração do sinal de entrada ocorrer através de um mecanismo não-linear. Um exemplo é o reconhecimento de voz.

(ii) Mapeamento de Entrada pela Saída: Após várias mudanças nos pesos sinápticos, a rede neural vai estabilizar-se e a resposta desejada para uma amostra de entrada deve ser obtida. Essa propriedade continuará valendo mesmo em situações nos quais a amostra de entrada é reaplicada em uma ordenação distinta. É desta forma que uma rede pode ser ensinada, construindo o mapeamento através de exemplos de entrada feitos para o problema considerado.

(iii) Adaptabilidade: Através de sua flexibilidade, as redes neurais são capazes de ajustarem-se em tempo real a mudanças em seus contextos de funcionamento através da modificação dos pesos sinápticos conectando seus neurônios, ou seja, podem ser retreinadas. Isso torna as redes neurais objetos de grande utilidade para classificação e controle adaptativo de padrões.

(iv) Resposta a Evidências: Redes Neurais podem ser projetadas para considerar a confiabilidade na seleção de um padrão, além de apenas qual padrão selecionar. Dessa forma, a rede pode evoluir para recusar padrões que sejam considerados duvidosos e aprimorar a performance da classificação de padrões pela rede.

(v) Informação Contextual: O próprio contexto da ordenação e estados das redes neurais representa a informação armazenada nele, já que todos os neurônios são afetados pelas atividades uns dos outros.

(vi) Tolerância a Falhas: Redes Neurais tem o seu desempenho reduzido de forma tênue mesmo sob condições desfavoráveis de atuação. Por ser um modelo distribuído de armazenamento de informação, é preciso um dano que seja muito extenso para que a resposta do sistema se degrade como um todo. A princípio, falhas catastróficas são menos comuns que em outros modelos, embora não sejam impossíveis.

(vii) Implementação em *Very-Large-Scale-Integration* (VLSI), ou Integração em Escala Muito Ampla em português, é uma tecnologia que permite obter comportamentos complexos hierarquicamente de uma rede neural que utiliza-la para sua implementação. O uso dessa tecnologia para redes neurais é possível graças a sua natureza paralela e de larga escala, potencializando sua velocidade na computação de determinados problemas.

(viii) Uniformidade de Análise e Projeto: Todos os projetos envolvendo Redes Neurais utilizam, de uma forma ou outra, os chamados neurônios. Isso possibilita o projeto de redes modulares que podem ser integradas homogeneamente em sistemas de aprendizado e a partilha de novos conhecimentos de Redes Neurais com outras aplicações que também usem essa mesma família de modelo de aprendizado.

(ix) Analogia Neurobiológica: Redes Neurais são baseadas na ideia de que o cérebro evidência que um processamento paralelo tolerante a falhas tem poder e velocidade, além de sua existência permitir concluir que tal processamento seja de construção viável.

Porém, as redes neurais sofrem de um problema chamado "dilema da estabilidade plasticidade", que afeta a efetividade de sua adaptabilidade. O dilema diz que o tempo principal de reação no sistema deve ser grande o suficiente para ignorar distúrbios ilegítimos, mas pequeno o bastante para reagir corretamente a mudanças consideráveis no contexto do sistema (HAYKIN, 2008). O tempo principal de reação precisa ser medido conforme esse dilema para alcançar o melhor modelo possível para a Rede Neural.

Esse dilema não é nada que dissuada contra a utilização das redes neurais, apenas um ponto a ser levado em consideração na criação das mesmas. O entendimento dos procedimentos apropriados a serem considerados na criação das redes neurais é necessário, pois o próximo tópico a ser explorado é um tipo de arquitetura de Redes Neurais chamada de Deep Neural Network(DNN) ou Rede Neural Profunda em

português. É uma rede com relacionamentos não-lineares complexos para a utilização de um modelo de aprendizado de máquina conhecido como *Deep Learning.*

2.3.1 *DEEP LEARNING*

Deep Learning possui várias definições, que podem ser resumidas da seguinte forma: É uma classe de algoritmos para aprendizado de máquina que é baseada no ensino de representações para modelar relacionamentos complexos dentro de dados. Utiliza múltiplas camadas de processamento não-linear de informações para extração e transformação de padrões. Pode ser um aprendizado supervisionado ou não-supervisionado. Tipicamente é implementada através de redes neurais artificiais. A metodologia dela é basear a aprendizagem em múltiplos níveis de representação e abstração existentes em uma hierarquia de características, de alto e baixo nível, que será chamada de Arquitetura Profunda (DENG; YU, 2014).

Deep Learning está na intersecção entre as seguintes áreas de pesquisa: redes neurais, inteligência artificial, modelagem gráfica, otimização, reconhecimento de padrões e processamento de dados. Sua popularidade cresce na medida que os processadores são aprimorados, a quantidade necessária de dados para treinamento aumenta e as investigações na área avançam. Há dois aspectos importantes para *Deep Learning* (DENG; YU, 2014):

- Modelos consistindo de múltiplas camadas aplicadas para estágios de processamento não linear de informação.
- Aprendizado supervisionado ou não-supervisionado em camadas sucessivamente mais abstratas.

Métodos de *Deep Learning* conseguem efetivamente explorar funções não-lineares complexas e compostas para aprender representações hierárquicas e distribuídas de características, além de fazerem uso efetivo de dados classificados e não-classificados. Pode-se categorizar a maior parte das arquiteturas de redes profundas e suas técnicas em três classes, dependendo das intenções de cada uma (DENG; YU, 2014):

- Para aprendizado não-supervisionado ou generativo: buscam capturar correlações de alta ordem em dados observáveis para análise de padrões, sintetizando seus propósitos e gerando classificações mesmo sem informações previamente disponíveis sobre os dados.

- Para aprendizado supervisionado: Também são chamadas de redes discriminativas. Providenciam diretamente o processo de discriminação para a classificação de padrões, geralmente através da caracterização de distribuições para classes existentes em dados visíveis. As classificações utilizadas estarão disponíveis em forma direta ou indireta.

- Para aprendizado híbrido: Também busca a discriminação, mas é auxiliada por métodos generativos ou não-supervisionados. É uma arquitetura profunda que utiliza as duas outras classes em conjunto.

As seguintes etapas são recomendadas na projeção de sistemas utilizando *Deep Learning* (GOODFELLOW *et al.*, 2016):

(i) Determinar os objetivos. Deve-se descobrir qual métrica de erro deve ser utilizada e qual deve ser o valor. Os objetivos e as métricas de erro devem ser baseadas no problema que a aplicação busca resolver.

(ii) Estabelecer estimativas do trabalho necessário para o projeto do sistema, incluindo quais serão as métricas utilizadas para avaliar a performance.

(iii) Instrumentar o sistema para determinar gargalos na desempenho. Diagnosticar quais componentes estão agindo pior do que o esperado e qual é a origem dos problemas.

(iv) Repetidamente fazer mudanças incrementadoras como obter dados novos, ajustar parâmetros ou mudar algoritmos, baseando-se em resultados obtidos durante a instrumentação do sistema.

Alguns parâmetros importantes para criar um sistema de redes neurais profundas são: a taxa de aprendizado, a quantidade de camadas escondidas, coeficiente de decaimento para peso, largura do núcleo de convolução(que controla o número de parâmetros no modelo), taxa de evasão das camadas e preenchimento implícito de zeros(serve para aumentar o tamanho das representações) (GOODFELLOW *et al.*, 2016).

Existem muitos tipos de redes neurais profundas, mas dentre as diferentes arquiteturas de *Deep Learning* existe uma de relevância para este trabalho chamada de *Autoenconder*. Essa arquitetura comumente utiliza uma abordagem de treino conhecida por *backpropagation*.

2.3.2 *BACKPROPAGATION*

A informação fluí através das redes de pré-alimentação quando elas aceitam uma entrada x e produzem uma saída Y. As entradas x proveem informações iniciais que se propagam pelas camadas escondidas até produzir Y. Eventualmente, esse processo produzirá um custo escalar(gradiente) J para as entradas x_i. Os algoritmos de *backpropagation* permitem que os gradientes sejam propagados reversalmente pela rede, começando pelo fim, para computa-los (GOODFELLOW *et al.*, 2016).

A motivação para o *backpropagation* é que computar os gradientes de forma direta pode ser computacionalmente caro. Algoritmos de *backpropagation* fazem isso usando um procedimento simples e pouco dispendioso. Além disso, o algoritmo geral de *backpropagation* foi inventado para evitar múltiplas computações das mesmas expressões secundárias em uma sequência de regras. Isso gerava aumento exponencial no tempo de execução (GOODFELLOW *et al.*, 2016).

De fato, o *backpropagation* refere-se apenas ao método para computar o gradiente, enquanto que algoritmo realizando o aprendizado é outro, como no caso do algoritmo da descida estocástica de gradiente. Implementações do *backpropagation* em software usualmente providenciam tanto as operações quanto os métodos necessários, assim usuários de bibliotecas de software para *deep learning* são capazes de utilizar esse método através de gráficos construídos utilizando operações comuns como multiplicação de matrizes, expoentes, logaritmos e outros (GOODFELLOW *et al.*, 2016).

Backpropagation é um método comum para treinar redes neurais. Para explica-lo, será apresentado um exemplo básico: uma rede neural com duas entradas, dois neurônios escondidos e dois de saídas. Os neurônios vão incluir vieses.

Na Figura 8, os números próximos de cada termo são seus valores. Os w_n são pesos iniciais, os que estão ao lado de cada b_n são para vieses, os i_n são entradas e o_n são as saídas do treino. O objetivo do *backpropagation* é otimizar os pesos de forma que a rede neural aprenda como mapear entradas arbitrarias para saídas.

Neste tutorial, são assumidas duas entradas: 0,05 e 0,10, com saídas que devem ser 0,01 e 0,99. Primeiro, verifica-se o que a rede neural retorna para os pesos e vieses da Figura 8 e as entradas mencionadas. Para isso, as entradas são alimentadas para a rede. É descoberta a entrada líquida para cada neurônio de camada escondida, usando uma função de ativação(neste caso, será a logística) para obter a saída do neurônio. O processo é repetido com os neurônios da camada de saída.

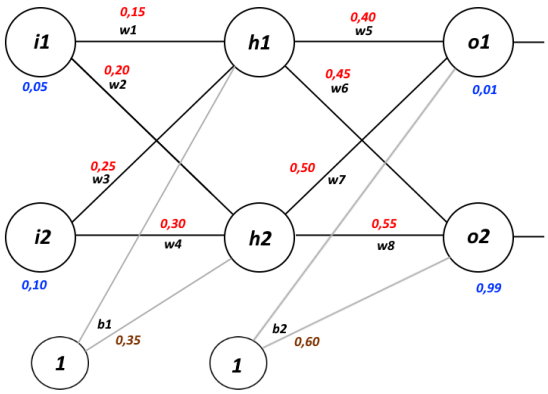

Figura 8 – Exemplo de Rede Neural para explicar *backpropagation*
Fonte: Autoria própria.

Para $h1$, o cálculo da entrada líquida é:

$$net_{h1} = w_1 * i_1 + w_2 * i_2 + b_1 * 1 \tag{3}$$

$$net_{h1} = 0,15 * 0,05 + 0,2 * 0,1 + 0,35 * 1 = 0,3755 \tag{4}$$

Em seguida, utiliza-se a função logística para obter a saída de $h1$:

$$out_{h1} = \frac{1}{1 + exp^{-net_{h1}}} = \frac{1}{1 + exp^{-0,3755}} = 0,593269992 \tag{5}$$

Fazendo o mesmo processo para $h2$:

$$out_{h2} = 0,596994378 \tag{6}$$

O processo é repetido para os neurônios de saída, usando a saída dos neurônios da camada escondida como entrada: A saída para $o1$ é:

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1 \tag{7}$$

$$net_{o1} = 0,4 * 0,593269992 + 0,45 * 0,596884378 + 0,6 * 1 = 1,105905967 \tag{8}$$

$$out_{o1} = \frac{1}{1 + exp^{-net_{o1}}} = \frac{1}{1 + exp^{-1,105905967}} = 0,75136507 \tag{9}$$

Realizando o mesmo processo para o_2:

$$out_{o2} = 0,772928465 \tag{10}$$

Em seguida, é calculado o erro para cada neurônio de saída usando a função de erro ao quadrado. Os erros encontrados são somados para obter o erro total.

$$E_{total} = \sum \frac{1}{2}(alvo - saida)^2 \tag{11}$$

O $\frac{1}{2}$ é incluído para que o expoente seja cancelado quando for realizada a diferenciação. O resultado é eventualmente multiplicado por uma taxa de aprendizado, portanto não importa que uma constante seja introduzida.

A saída alvo de o_1 é 0,01, mas a saída da rede neural é 0,75136507. O erro será:

$$E_{o_1} = \frac{1}{2}(0,01 - 0,75136507)^2 = 0,274811083 \tag{12}$$

E para o_2, com alvo sendo 0,99:

$$E_{o_2} = 0,023560026 \tag{13}$$

O erro total da rede neural é a soma desses erros:

$$E_{total} = E_{o_1} + E_{o_2} = 0,274811083 + 0,023560026 = 0,298371109 \tag{14}$$

Com o erro total obtido, o objetivo com *backpropagation* é atualizar cada peso da rede para que a saída atual fique cada vez mais perto da saída alvo, minimizando o erro para cada neurônio de saída e para a rede inteira. O método para a atualização dos pesos é ilustrado pela Figura 9 (MAZUR, 2015).

Explicando o método: considere $w5$. Deseja-se saber como uma mudança no peso $w5$ afetaria o seu erro total $\frac{\delta E_{total}}{\delta w5}$, que é lido como uma derivada parcial de E_{total} em relação a $w5$ (ou gradiente com relação a $w5$).

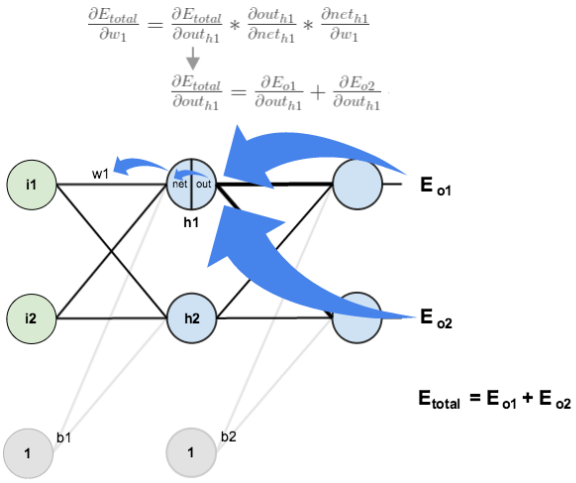

Figura 9 – O passo para trás feito pelo *backpropagation*, atualizando o peso $w1$
Fonte: Mazur (2015).

Aplicando a regra da cadeia para $w5$:

$$\frac{\delta E_{total}}{\delta w5} = \frac{\delta E_{total}}{\delta out_{o1}} * \frac{\delta out_{o1}}{\delta net_{o1}} * \frac{\delta net_{o1}}{\delta w5} \tag{15}$$

Para descobrir como o erro total muda com relação a saída:

$$E_{total} = E_{o_1} + E_{o_2} \tag{16}$$

$$E_{total} = \frac{1}{2}(alvo_{o1} - saida_{o1})^2 + \frac{1}{2}(alvo_{o2} - saida_{o2})^2 \tag{17}$$

$$\frac{\delta E_{total}}{\delta out_{o1}} = -2 * \frac{1}{2}(alvo_{o1} - saida_{o1}) \tag{18}$$

$$\frac{\delta E_{total}}{\delta out_{o1}} = -(alvo_{o1} - saida_{o1}) = -(0,01 - 0,75136507) = 0,74136507 \tag{19}$$

A seguir, para saber como a saída de $o1$ muda com relação a entrada líquida é feita a derivada parcial da função logística:

$$out_{o1} = \frac{1}{1 + exp^{-net_{o1}}} \tag{20}$$

$$\frac{\delta out_{o1}}{\delta net_{o1}} = out(1 - out_{o1}) = 0,75136507(1 - 0,75136507) = 0,186815602 \tag{21}$$

Finalmente, como a entrada liquida de $o1$ muda com relação a $w5$:

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1 \tag{22}$$

$$\frac{\delta net_{o1}}{\delta w5} = out_{h1} = 0,593269992 \tag{23}$$

Com esses três resultados, encontra-se $\frac{\delta E_{total}}{\delta w5}$.

$$\frac{\delta E_{total}}{\delta w5} = -(alvo_{o1} - saida_{o1}) * out(1 - out_{o1}) * out_{h1} \tag{24}$$

$$\frac{\delta E_{total}}{\delta w5} = 0,74136507 * 0,186815602 * 0,593269992 = 0,082167041 \tag{25}$$

Para diminuir o erro, pode-se subtrair esse valor do peso atual (opcionalmente multiplicado por alguma taxa de aprendizado η que neste caso será 0,5):

$$w_5' = w_5 - \eta * \frac{\delta E_{total}}{\delta w5} = 0,4 - 0,5 * 0,082167041 = 0,35891648 \tag{26}$$

Esse processo pode ser repetido para conseguir os novos pesos de $w2$, $w3$ e $w4$:

$$w_6' = 0,408666186 w_7' = 0,511301270 w_8' = 0,561370121 \tag{27}$$

Atualizações na rede neural ocorrem somente depois da obtenção de novos pesos para os neurônios de camadas escondidas, ou seja, agora são calculados novos valores para $w1$, $w2$, $w3$ e $w4$.

O processo será similar ao usado para a camada de saída, mas a diferença é que cada neurônio na camada escondida contribui com múltiplos neurônios de saída, ou seja, com o erro também.

$$\frac{\delta E_{total}}{\delta w1} = \frac{\delta E_{total}}{\delta out_{h1}} * \frac{\delta out_{h1}}{\delta net_{h1}} * \frac{\delta net_{h1}}{\delta w1} \tag{28}$$

Sabe-se que out_{h1} afeta tanto out_{o1} e out_{o2}. Portanto, $\frac{\delta E_{total}}{\delta out_{h1}}$ precisa levar em consideração os efeitos de ambos os neurônios de saída:

$$\frac{\delta E_{total}}{\delta out_{h1}} = \frac{\delta E_{o1}}{\delta out_{h1}} + \frac{\delta E_{o2}}{\delta out_{h1}} \tag{29}$$

O cálculo de $\frac{\delta E_{o1}}{\delta out_{h1}}$ e $\frac{\delta E_{o2}}{\delta out_{h1}}$ podem ser feitos com valores que foram calculados anteriormente:

$$\frac{\delta E_{o1}}{\delta out_{h1}} = \frac{\delta E_{o1}}{\delta net_{o1}} * \frac{\delta net_{o1}}{\delta out_{h1}} = (0,74136507 * 0,186815602) * \frac{\delta net_{o1}}{\delta out_{h1}} \tag{30}$$

$$net_{o1} = w5 * out_{h1} + w6 * out_{h2} + b2 * 1 \tag{31}$$

$$\frac{\delta net_{o1}}{\delta out_{h1}} = w5 = 0,40 \tag{32}$$

$$\frac{\delta E_{o1}}{\delta out_{h1}} = 0,138498562 * 0,40 = 0,055399425 \tag{33}$$

Para o $\frac{\delta E_{o2}}{\delta out_{h1}}$:

$$\frac{\delta E_{o2}}{\delta out_{h1}} = -0,019049119 \tag{34}$$

Portanto:

$$\frac{\delta E_{total}}{\delta out_{h1}} = 0,055399425 - 0,019049119 = 0,036350306 \tag{35}$$

Aogra é preciso descobrir $\frac{\delta out_{h1}}{\delta net_{h1}}$ e então $\frac{\delta net_{h1}}{\delta w}$ para cada peso:

$$out_{h1} = \frac{1}{1 + exp^{-net_{h1}}} \tag{36}$$

$$\frac{\delta out_{h1}}{\delta net_{h1}} = out_{h1}(1 - out_{h1}) = 0,59326999(1 - 0,59326999) = 0,241300709 \tag{37}$$

A derivada parcial da entrada liquida $h1$ com relação a $w1$ é calculada da mesma forma que foi feita para o neurônio de saída:

$$net_{h1} = w_1 * i_1 + w_3 * i_2 + b_1 * 1 \tag{38}$$

$$\frac{\delta net_{h1}}{\delta w_1} = i_1 = 0,05 \tag{39}$$

Dessa forma, $\frac{\delta E_{total}}{\delta w1}$ será:

$$\frac{\delta E_{total}}{\delta w1} = 0,036350306 * 0,241300709 * 0,05 = 0,000438568 \tag{40}$$

Agora pode-se atualizar $w1$:

$$w_1' = w_1 - \eta * \frac{\delta E_{total}}{\delta w1} = 0,15 - 0,5 * 0,000438568 = 0,149780716 \tag{41}$$

Repetindo para $w2$, $w3$ e $w4$:

$$w_2' = 0,19956143 w_3' = 0,24975114 w_4' = 0,29950299 \tag{42}$$

Todos os pesos foram atualizados. Quando as entradas 0,05 e 0,1 foram alimentadas, o erro da rede era 0,298371109. Depois da primeira rodada de *backpropagation*, o erro total caiu para 0,291027924. Pode não ser muito, mas depois que o processo for repetido 10000 vezes, por exemplo, o erro cairá para 0,0000351085. Quando alimentarmos as entradas 0,05 e 0,1 novamente, os neurônios de saída vão gerar 0,015912196 (versus 0,01 do alvo) e 0,984065734 (versus 0,99 do alvo).

2.3.3 *AUTOENCODER*

Um *Autoencoder* é um tipo especial de rede neural que é treinada para que o vetor de entradas tenha a mesma dimensionalidade dos vetores de saída. Geralmente utilizada para aprendizado não-supervisionado, gerando uma representação ou codificação efetiva para dados de entrada nas camadas escondidas. Quando o seu número de camadas escondidas é maior que um, então o *Autoencoder* é considerado uma rede neural profunda. Comparada as dimensões das entradas, as camadas escondi-

das podem ser pequenas se o objetivo for comprimir características ou grandes se o propósito for mapeá-las em espaços de maior dimensão (DENG; YU, 2014).

Um *Autoencoder* tipicamente tem uma camada de entrada que representa os dados originais ou as características de entrada, uma ou mais camadas escondidas que representam as características transformadas, e uma camada de saída que compatibiliza com a de entrada na reconstrução das características inicias (DENG; YU, 2014). É um método não-linear para extrair características que não utiliza classificação, sendo que a extração foca em conservar e melhorar a representação das informações ao invés de classificar processos, embora algumas vezes os dois objetivos estejam relacionados (DENG; YU, 2014).

Treinados comumente com diferentes variações do *backpropagation*, tipicamente pelo método do gradiente estocástico. Um problema de utilizar esse método é que o erro, quando propagado para muitas camadas escondidas, se torna inefetivo pois fica minusculo demais. Isso resulta em aprendizagem lenta e reconstruções inúteis, especialmente se dados de treino forem limitados. O problema pode ser aliviado através do pré-treino de cada camada de *Autoencoder* simples (DENG; YU, 2014).

Internamente, sua forma clássica possui uma camada escondida h que descreve um código usado para representar a entrada. Sua rede consiste de duas partes: a função codificadora $h = f(x)$ e a decodificadora que produz a reconstrução $r = g(h)$. A arquitetura está presente na Figura 10 (GOODFELLOW *et al.*, 2016).

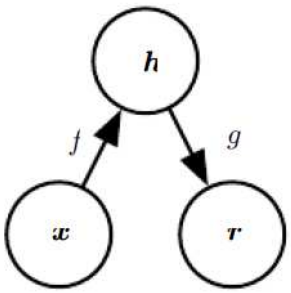

Figura 10 – Estrutura geral de um *autoencoder*, mapeando uma entrada x para uma saída reconstruída r através de uma representação interna ou código h
Fonte: Goodfellow (2016, p. 503).

Autoencoders são projetados para serem capazes de realizarem uma reconstrução perfeita da entrada, embora fatores como dados de treino ou número de camadas escondidas no *Autoencoder* possam limitar a sua capacidade de reconstrução da entrada. Também é possível projetar um *Autoencoder* para reconstruir apenas aspectos da entrada. O modelo geral de *Autoencoder* foca em aprender propriedades

úteis das informações, priorizando quais aspectos da entrada devem ser copiados nas reconstruções (GOODFELLOW *et al.*, 2016).

Se o codificador de um *Autoencoder* for determinístico, então pode ser considerado como uma rede neural pré-alimentada. Funções de perda e tipos de unidades de saída utilizadas nessas redes também podem ser usadas para os *Autoencoders*. Porém, podem utilizar algoritmos adicionais para seu processo de treinamento além do *backpropagation* (GOODFELLOW *et al.*, 2016).

Alguns tipos de *Autoencoders* são (GOODFELLOW *et al.*, 2016):

- Incompletos: São *Autoencoders* que constringem a dimensão de saída para que seja menor que a de entrada, forçando o *Autoencoder* a capturar as características mais proeminentes de dados treinadores. É implementado minimizando a função de perda do processo de aprendizado.

- Regularizados: Permitem escolher a dimensão de código e a capacidade dos codificadores e decodificadores baseando-se na complexidade da distribuição que vai ser modelada neste *Autoencoder*. Ao invés de limitar a capacidade do modelo, mantendo o codificador e decodificador pequenos, eles podem usar uma função de perda para induzir o modelo em ter outras propriedades além da habilidade de reconstruir a entrada na saída. Podem ser não-lineares e ultrapassar as dimensões de entrada, mas ainda aprendem informações úteis sobre a distribuição dos dados mesmo com o modelo sendo tão grande que acabe aprendendo resultados triviais.

- Esparsos: Tipicamente utilizados para aprender características de outros processos, como os de classificação. Eles respondem apenas para características de estatísticas especiais no conjunto de dados que foi usado para treino, ao invés de simplesmente agirem como funções de identificação.

- Descorruptores: Recebe dados corrompidos ou com ruído como entrada e são treinados para retornar a forma original dos dados na saída. Para o seu treinamento, amostras de dados que foram propositalmente corrompidos são dadas como entrada. Desta forma, estimando através dos dados corrompidos e originais utilizados no treino o *Autoencoder* aprende uma forma de reconstrução.

- Contrativos: Treinados para resistirem a pertubações na entrada, induzindo o mapeamento de pontos de entrada com os de saída para que sejam vizinhos

e menores. Isso cria uma contração local, com qualquer ruído de um ponto de treino x sendo mapeados perto de uma saída próxima $f(x)$. Um problema é que para *Autoencoders* profundos(de várias camadas de neurônios) ele torna o processo de computação consideravelmente custoso, além disso pode obter resultados inúteis se não existir alguma escala no decodificador.

Os *Autoencoders* foram aplicados com sucesso para problemas de redução de dimensionalidade e extração de informações em processos. Representações com menos dimensões aprimoram o desempenho de muitas operações como as de classificação. Modelos com menor espaço consumem menos tempo de execução e memória. Processos que extraem informações também beneficiam-se de dimensões menores, realizando buscas mais eficientes. Um exemplo disso são métodos de busca em banco de dados que utilizam tabela *hash*, mapeando códigos binários para entradas. Um *Autoencoder* pode aprender uma função *hash* para mapeamento (GOODFELLOW *et al.*, 2016).

Os *Autoencoders* são utilizados neste trabalho para um sistema de detecção de códigos maliciosos, fazendo parte do processo de extração de características de malwares. A arquitetura de um *Autoencoder* é vista na imagem 11:

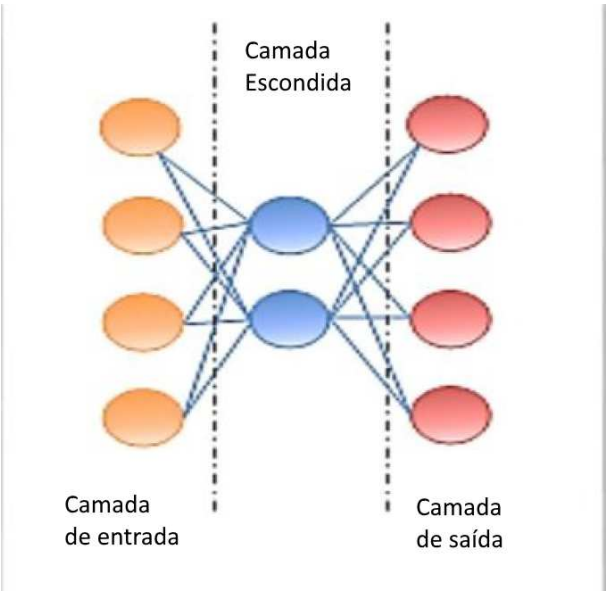

Figura 11 – Representação da arquitetura de um *Autoencoder*
Fonte: Traduzido de Zaccone (2017, p. 98).

A Figura 11 apresenta um *Autoencoder* tradicional: duas camadas(entrada e saída) com uma camada escondida. O mesmo número de neurônios existe nas camadas de entrada e saída. Quatro números de valores reais são alimentados ao *Autoencoder*, que os comprime com o codificador em dois valores reais na camada

escondida. Usando estes dois valores, o decodificador tenta reconstruir os quatro números reais que foram alimentados como entrada na rede.

Em aplicações reais, vão existir mais camadas escondidas entre a entrada e saída.

2.4 SEGURANÇA COMPUTACIONAL

A Segurança Computacional é a proteção de sistemas computacionais e seus dados armazenados contra acesso não-autorizado através do processo de prevenção, detecção e remoção de ameaças dentro de um sistema. A Segurança Computacional está presente em várias redes utilizadas no dia-a-dia da população. Acesso online de bancos, compras em mercados virtuais, compra de passagens de viagem e outros. Os objetivos da Segurança Computacional são (EASTTOM, 2012):

(i) Identificar as maiores ameaças as redes de computadores;

(ii) Determinar a probabilidade de um ataque;

(iii) Rotular e estudar terminologias como *cracker*, *malware*, *firewall* e autenticação;

(iv) Comparar e diferenciar diferentes métodos de proteção de rede;

(v) Permitir o uso seguro de recursos online.

Existem vários benefícios em utilizar aprendizado de Máquina na área de segurança computacional. O primeiro é que ao invés de foco em conjuntos de regras e assinaturas, o aprendizado de máquina permite uma abordagem de adaptação continua baseada no computador sendo ensinado diretamente de dados obtidos. Outra vantagem é que problemas de Classificação, Predição Quantitativa, Inferência, Exploração e Descoberta de informações relevantes para a segurança dos computadores podem ser resolvidos pelos métodos de aprendizados de máquina. Isso vale para os supervisionados, semi-supervisionados e não-supervisionados. Os três são capazes de solucionar diferentes questões da área de Segurança Computacional (JACOBS; RUDIS, 2014).

Os problemas de Segurança Computacional estão presentes em diversas áreas técnicas, como: a Internet das Coisas(*Internet of Things* (IoT) no inglês), Investigação Forense, Aplicações, Criptografia e Criptoanálise, Computação em Nuvem, Proteção

de Software, Sistemas Distribuídos, Suporte de Hardware e outras (BASTOS, 2016). A Internet das Coisas é uma rede virtual que permite o controle de objetos físicos, cujas ameaças mais importantes são ataques de invasores que desejem controlar os objetos remotamente. Ameaças para Investigação Forense são programas ofuscadores que desejem atrapalhar o trabalho de investigadores em um computador apreendido. Criptografia busca proteger a privacidade dos dados de observadores alheios e precisa impedir a quebra do seu sistema codificador, enquanto Criptoanálise busca desvendar dados codificados e também são afetados por metodologias ofuscadoras. Computação em Nuvem precisa garantir a consistência e privacidade dos dados existentes na rede. Proteção de Software envolve impedir a utilização ilícita de um programa, como em casos de cópias não-comerciais sendo distribuídas e utilizadas por usuários finais sem pagar o preço do software.

As outras áreas citadas possuem ameaças mais diversas como roubo de dados, danificação dos dados e controle ilícito dos sistemas. Um tipo de ameaça a Segurança Computacional é o Malware, um software malicioso que é amplamente utilizado como instrumento de realização das outras ameaças que foram apresentadas. Este trabalho lida com a questão de detecção dos Malwares, portanto está dentro da área de Segurança Computacional.

2.5 MALWARES

Um código malicioso, também conhecido como (Malware), é um programa que secretamente se insere em outro executável ou em um dispositivo como na Figura 12. Seu propósito é servir como ferramenta para atividades criminosas como roubar ou destruir dados de empresas. Malwares comprometem a confidenciabilidade, integridade e disponibilidade de sistemas operacionais e seus aplicativos. É a ameaça externa mais comum para computadores, causando prejuízos generalizados em muitas organizações que são obrigadas a dispensarem esforços em mecanismos extensivos de recuperação (SOUPPAYA; SCARFONE, 2013).

2.5.1 TIPOS COMUNS DE MALWARES

As seguintes categorias são algumas clássicas formas de malware (SOUPPAYA; SCARFONE, 2013):

- Vírus: É um programa capaz de realizar autorreplicação, inserindo cópias

Figura 12 – **Malware para Android**
Fonte: Malwarebytes (2017, p. 7).

de si mesmo em programas ou arquivos de dados. Geralmente desencadeados através de interação do usuário, como abrir um arquivo ou executar um programa.

- *Worm*: É um programa que também replica a si mesmo, mas é autônomo e normalmente iniciado sem intervenção de um usuário.

- *Trojan Horses*: Outro programa autônomo, mas não-replicante. Pretende ser benigno para esconder um propósito malicioso. Substituem arquivos legítimos com versões maliciosas ou adicionam novas arquivos maliciosos nos hospedeiros como ferramentas para atacar o sistema.

- Código Malicioso Móvel: Programa autônomo com propósitos maliciosos que é transmitido de um hospedeiro remoto para ser executado no novo sistema que o hospeda, tipicamente sem interação do usuário.

Um novo tipo de malware que surgiu recentemente é o chamado Ransomware. Focado em atingir negócios. O objetivo deste ataque é sequestrar dados do sistema da empresa, criptografando-os e exigindo dinheiro em troca da chave que decodifica-os (MALWAREBYTES, 2017).

Ransomware difere dos malwares tradicionais por ser de fácil compra por hackers de baixa habilidade. São vendidos em forma de kits que simplificam a utilização deles para atividades ilegais. Somente em 2016 foram catalogadas 400 variantes de ransomwares que foram criados apenas por um grupo novo de criminosos (MALWAREBYTES, 2017).

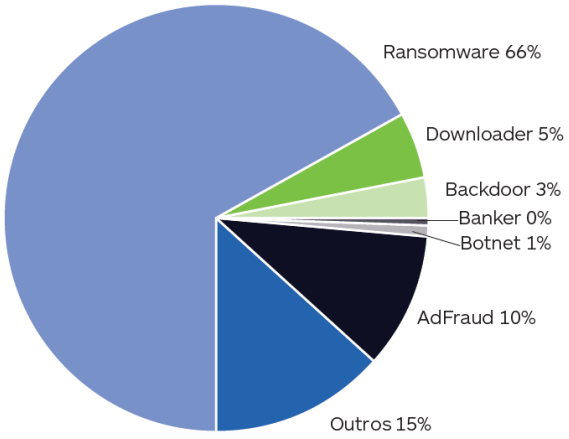

Figura 13 – Distribuição dos tipos de malwares em Novembro de 2016
Fonte: Adaptado de Malwarebytes (2017, p. 2).

Outra forma de classificação é baseando-se na extensão da abrangência desejada pelos criadores dos códigos maliciosos. Alguns malwares são projetados para ataques em massa ou para alvos específicos. Geralmente, malwares feitos para alvos específicos são ameaças maiores as redes porque produtos de segurança computacional provavelmente fornecerão proteção menor contra eles, já que esses malwares não estão distribuídos em larga escala infectando vários computadores e portanto são menos conhecidos (SIKORSKI; HONIG, 2012).

Novos malwares que não se encaixam nas categorias acima mencionadas continuam a surgir, alguns ataques podem até combinar múltiplos métodos de infecção ou transmissão. Deve-se evitar dispensar recursos em categorizar cada incidente com malwares baseando-se nas categorias citadas, devido a continua evolução de sua complexidade. Para análise de malwares, o foco é em detectar características chaves dos programas maliciosos (SIKORSKI; HONIG, 2012).

2.5.2 FERRAMENTAS DE ATAQUE

Malwares utilizam ferramentas que permitem que criminosos consigam infectar os sistemas computacionais e seus dados, tenham acesso não-autorizado ou fa-

çam mais ataques ao sistema. Alguns tipos de ferramentas são (SOUPPAYA; SCARFONE, 2013):

- *Backdoors*: Permitem que um invasor realize uma série de ações no hospedeiro, como adquirir senhas ou executar comandos arbitrários como atacar outros sistemas.

- *Keyloggers*: Monitora e recorda o uso do teclado no computador, armazenando as teclas digitadas e depois transfere os dados para o criminoso atacando o sistema. Comumente usado para roubo de senhas.

- *Rootkits*: Coletânea de arquivos instalados no hospedeiro para alterar a sua funcionalidade padrão de forma indetectável, fazendo com que o sistema hospedeiro realize atividades prejudiciais aos seus legítimos utilizadores.

- *Plug-Ins*: de Navegadores: Alguns deles podem ser maliciosos. Por trás da atividade pelo qual foram instalados no navegador podem estar monitorando-o.

- *Toolkits*: Contem diferentes tipos de utilidades e *scripts* que podem ser usados para sondar e atacar sistemas, como decifradores de senhas ou escaneadores de portas vulneráveis.

Ferramentas de ataque podem ser feitas e vendidas por outros criminosos para facilitar as atividades de invasores de sistemas e permitirem até sua customização, amplificando ainda mais o problema de invasões em sistemas (SOUPPAYA; SCARFONE, 2013).

O comportamento das ferramentas de ataque geram características que podem ser úteis para detecção dos malwares que esses métodos procuram instalar. Malwares sofisticados vão precisar de análise extensiva para serem detectados, ou seja, um sistema de aprendizado utilizando *Deep Learning* seria útil já que há uma grande quantidade de características envolvidas no processo de detecção.

2.6 EXTRAÇÃO DE CARACTERÍSTICAS

Existem dois aspectos na extração de características:

- Construção de Características.

- Seleção de Características.

Extração de Características é feita para modelar uma representação de dados. Esse conjunto de informações vai representar um número fixo de características que podem ser binárias, categóricas ou contínuas. Característica é equivalente a uma variável de entrada e são específicas para a área do problema em que estão contextualizadas, além de dependerem de medições disponíveis. A entrada de dados é utilizada para construir as características extraídas deles pelo sistema de aprendizado (GUYON; ELISSEEFF, 2006).

2.6.1 CONSTRUÇÃO DE CARACTERÍSTICAS

A experiência humana é necessária para converter dados não-processados em configurações de características uteis e pode ser complementada por métodos que automaticamente constroem conjuntos de características. Isso significa que a construção de características pode ser integrada no processo de modelagem do sistema de aprendizado. Um exemplo são as camadas escondidas de redes neurais artificiais que computam representações internas análogas a características construídas (GUYON; ELISSEEFF, 2006), como ilustrado na Figura 14.

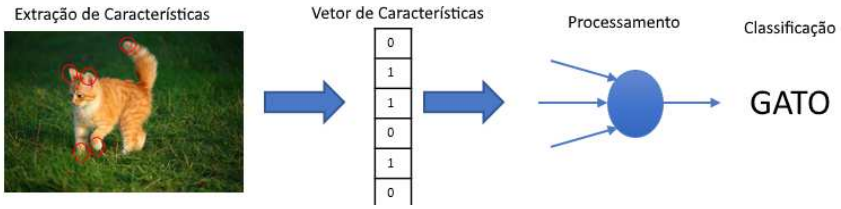

Figura 14 – Exemplo de construção de características, baseada em atributos comuns nas fotos contendo uma classe de animal

Fonte: Autoria própria.

Outra abordagem para a construção de características é o pré-processamento. Seja x um vetor de características com dimensão n, com x_i sendo uma característica, x = *[*x_1,x_2,...x_n*]* e x' o vetor de características transformadas de uma dimensão n'. Transformações de pré-processamento para o vetor podem conter as seguintes etapas (GUYON; ELISSEEFF, 2006):

- *Padronização*: características podem ter diferentes escalas mesmo para objetos comparáveis. Por exemplo, x_1 sendo comprimento em metros e x_2 sendo altura em centímetros. Operações entre os dois necessitariam antes de escalonamento e centralização desses dados.

- *Normalização*: Normalizar um dado distribuído em todas as características no vetor para eliminar essa dependência.

- *Aprimoramento de sinal*: Através de filtros para eliminar ruídos ou suavizar o sinal.

- *Dimensionamento dos dados*: quando a dimensionalidade das informações é alta, algumas técnicas podem ser usadas para projetar e integrar os dados em espaços de dimensões menores.

- *Discretização de características*: Alguns algoritmos não trabalham eficientemente com dados contínuos e precisam que esses valores sejam discretizados em um conjunto finito. Além disso, essa etapa pode simplificar a descrição de dados e melhorar sua compreensão.

É importante que informações não sejam perdidas durante as etapas de construção de características, mesmo que isso resulte em adicionar informações que podem não ser tão úteis. Como exemplos para ilustrar esse ponto: o diagnóstico médico que precisa de tantas informações sobre o paciente que forem possíveis de extrair, ou a própria detecção de códigos maliciosos que é o objetivo deste trabalho. Isso significa ter que aumentar a dimensionalidade dos padrões extraídos e correr o risco de submergir informações importantes em uma pilha de dados irrelevantes. Neste caso, para saber quais características são relevantes ou informativas é preciso utilizar métodos de seleção de características.

2.6.2 SELEÇÃO DE CARACTERÍSTICAS

Existem três itens relevantes para a seleção de características (GUYON; ELISSEEFF, 2006):

- Geração de subconjunto de características(ou estratégia de busca): envolve métodos de busca como os estocásticos ou as heurísticas, além dos de eliminação como *backward elimination*.

- Definição dos critérios de avaliação: Relevância em contexto e individual das características, além da relevância do próprio subconjunto de características.

- Estimação dos critérios de avaliação: testes estatísticos e limites de atuação dos critérios.

Além de encontrar informações importantes para o modelo de aprendizado, a seleção de características pode ter outras motivações como redução de dados e do conjunto de características, aumentar a velocidade do algoritmo, aprimorar a performance do sistema de aprendizado e entender os dados através de sua visualização. Métodos de seleção de características incluem filtros, coeficientes de correlação, testes clássicos de estatística como o T-test e empacotamento. Um aspecto crítico de seleção da características é determinar corretamente a qualidade das que são selecionadas. Isso inclui teste de hipóteses, validação cruzada(dividir dados de treino em duas partes: uma para treino e outra pra validação da performance do aprendizado) e design experimental(quantas amostras de treino são necessárias para solucionar o problema de seleção de características) (GUYON; ELISSEEFF, 2006).

Construção de características e seleção das mesmas são processos que podem ser feitos de forma simultânea, como ocorre no modelo utilizando Rede Neural Profunda que é abordado neste projeto. O sistema de aprendizado explorado vai extrair sequências de chamadas da API do Windows e seleciona-las como características dos códigos maliciosos.

2.6.3 *API CALLS*

Uma *Application Programming Interface* (API), ou Inferface de Programação de Aplicações em português, é um conjunto de símbolos que são exportados e disponibilizados para os usuários de uma biblioteca de códigos escreverem seus aplicativos. O modelo das APIs pode ser considerado a parte mais crítica no projeto da biblioteca porque afeta os projetos de aplicativos que serão construídos baseando-se nele (BLANCHETTE, 2008).

É possível detectar atividades maliciosas de malwares e caracteriza-los através da sequência de chamadas da (Win-API). Diferentes tipos de códigos maliciosos irão ter determinados padrões de utilização da API, o que significa que é possível categoriza-los através da identificação e extração de seus padrões de chamadas de sequência no Win-API para criar o vetor de características necessário para o método de detecção (GUPTA *et al.*, 2016).

Para utilizar eficientemente as *API Calls* para extração de vetores de características é preciso um grande número de amostras de Malware e também mais categorias do que aquelas que estão presentes na Figura 15. Além disso, vale lembrar que esse método de extração só deve ser considerado válido para o sistema operacional

Atividade Maliciosa	PADRÃO NA API
Key Logger	(FindWindowA, ShowWindow, GetAsyncKeyState) (SetWindowsHookEx, RegisterHotKey, GetMessage,UnhookWindowsHookEx)
Screen Capture	(GetDC, GetWindowDC), CreateCompatibleDC, CreateCompatibleBitmap, SelectObject, BitBlt, WriteFile
Antidebugging	(IsDebuggerPresent, CheckRemoteDebuggerPresent, OutputDebugStringA, OutputDebugStringW)
Downloader	URLDownloadToFile, (WinExec,ShellExecute)
DLL Injection	OpenProcess, VirtualAllocEx, WriteProcessMemory, CreateRemoteThread
Dropper	FindResource, LoadResource, SizeOfResource

Figura 15 – Alguns padrões maliciosos na API do Windows
Fonte: Adaptado de Gupta (2016, p. 7).

Windows. Outros sistemas operacionais não estão inclusos neste contexto (GUPTA *et al.*, 2016).

As *API Calls* podem ser obtidas através de programas extratores, que monitoram as chamadas de funções da API do Windows feitos por executáveis. Um desses programas é o WinAPIOverride64, que permite observar as chamadas de funções que foram feitas e a sequência em que ocorreram (POTIER, 2018). A Figura 16 ilustra um exemplo do funcionamento desse programa. As sequências de chamadas obtidas são os dados utilizados para o treinamento dos classificadores e do *Autoencoder* construído neste trabalho.

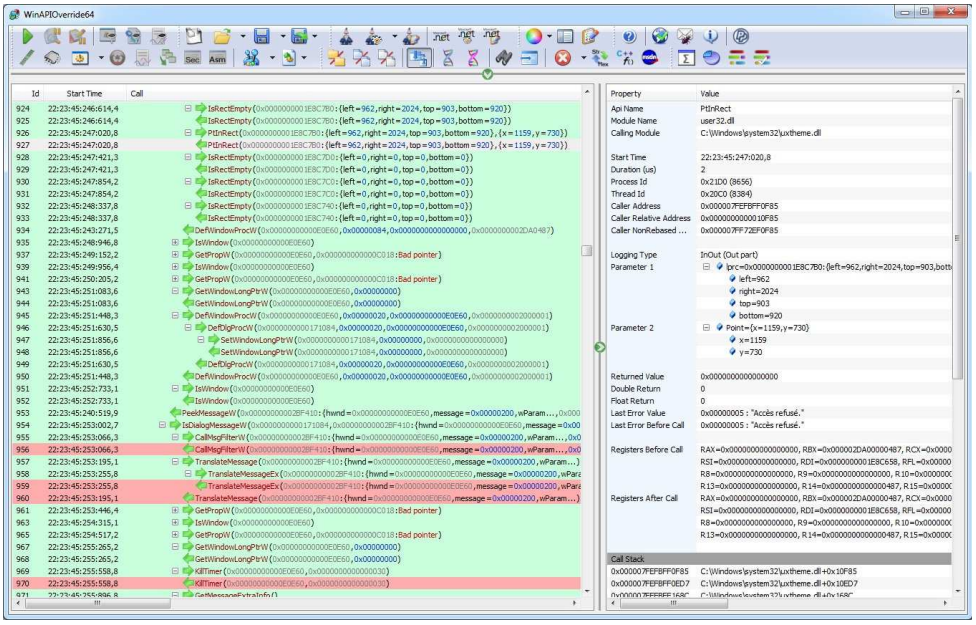

Figura 16 – WinAPIOverride64, uma ferramenta de extração de *API Calls*
Fonte: Potier (2018).

2.7 TENSORFLOW

O TensorFlow é um fornecedor de múltiplas APIs de código aberto. A API de maior baixo nível é a TensorFlow Core: que permite controle completo pelo programador e é recomendada para pesquisadores de aprendizado de máquina. As APIs de alto nível são feitas usando o TensorFlow Core como base e são de entendimento menos complexo que o TensorFlow Core, facilitando e padronizando tarefas repetitivas. Um exemplo é a API chamada *tf.estimator*, que ajuda a gerenciar conjuntos de dados, estimadores, treinamento e inferência. TensorFlow Core é a API que usa uma unidade central de dados denominada "tensor". Um tensor é um conjunto de valores em forma de matriz com n dimensões, sendo n o seu *rank* (ZADEH; RAMSUNDAR, 2017). Exemplos:

- 3 (Um tensor de rank 0, ou seja, uma escalar)
- [1 2 3] (Um tensor de rank 1, ou seja, um vetor)
- [[1 2 3], [4 5 6]] (Um tensor de rank 2, ou seja, uma matriz do tipo 2x3)
- [[[1 2 3]] , [[7 8 9]]] (Um tensor de rank 3)

Os programas do TensorFlow Core consistem de duas partes: Construir um grafo computacional e Rodar o grafo computacional. Um grafo computacional é uma série de operações do TensorFlow organizadas em um grafo de nós. Cada nó pega zero ou mais tensores como entrada e produzem um tensor de saída. Nós podem ser Tensores ou operações. Alguns nós podem ser constantes, e como todas as constantes do TensorFlow ele apenas armazena um valor de saída e não recebe entradas (ZADEH; RAMSUNDAR, 2017). Exemplo:

```
node1 = tf.constant(3.0, dtype=tf.float32)
node2 = tf.constant(4.0)
sess = tf.Session()
print(sess.run([node1, node2]))
```

O resultado: [3.0, 4.0]

Agora, se for utilizado um nó de operação:

```
node3 = tf.add(node1, node2)
```

```
print("node3:", node3)
print("sess.run(node3):", sess.run(node3))
```

O resultado dos comandos acima é:

```
node3: Tensor("Add:0", shape=(), dtype=float32)
sess.run(node3): 7.0
```

Para facilitar a visualização, o TensorFlow possui uma utilidade chamada TensorBoard que pode mostrar uma figura do grafo computacional, conforme visto na Figura 17

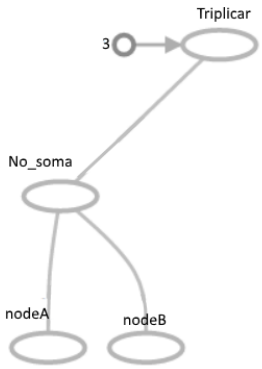

Figura 17 – **Exemplo de figura criada pelo TensorBoard**
Fonte: Autoria própria.

Um nó de um grafo pode receber uma operação chamada *placeholder*. Um *placeholder* é um espaço reservado para um tipo de valor que será recebido na entrada em algum momento. Em código, pode ser descrito como: tf.placeholder(*valor de espaço reservado*). Para avaliar o grafo, é possível dar um valor de entrada para os nós *placeholders* na execução da sessão. O mesmo pode ser feito com vetores. Em aprendizado de máquina se é tipicamente utilizado modelos com várias entradas arbitrarias, mas para ele ser treinável é preciso modificar o grafo para conseguir novas saídas com a mesma entrada. Operações chamadas *Variables* permitem que sejam adicionados parâmetros treináveis a um grafo. Elas são construídas com um tipo e um valor inicial: tf.Variable([*valor*], dtype=*tipo*) (ZADEH; RAMSUNDAR, 2017).

Esses nós variáveis não são inicializados assim que são declarados. Para serem inicializados é preciso que uma operação especial seja explicitamente chamada, conhecida como *tf.global_variables_initializer* e seguida de *sess.run(init)*. Utilizando nós variáveis em conjunto de placeholders é possível criar diversos modelos, que podem ser testados seguindo determinados parâmetros para descobrir sua eficiência. Em sistemas supervisionados isso significa treinar o modelo até que a função de perda

entregue um valor igual ou próximo de zero(ou de perda aceitável). Pode-se utilizar otimizadores da API do *tf.train* para esse propósito (ZADEH; RAMSUNDAR, 2017).

Os otimizadores do tf.train simplificam a criação de código utilizando TensorFLow, mas modelos mais complexos podem necessitar de mais código. O TensorFlow provê abstrações de alto nível para padrões, estruturas e funcionalidades comuns. E ainda, não é preciso ficar preso em modelos predefinidos das APIs. É possível criar modelos customizados dessas APIs dentro do seu código utilizando conhecimento de baixo nível da TensorFlow Core (ZADEH; RAMSUNDAR, 2017).

2.7.1 IMPLEMENTANDO UM *AUTOENCODER* COM O TENSORFLOW

Este trabalho vai utilizar o TensorFlow para implementar um *autoencoder* que utilizará um sistema *Deep Learning* para otimizar a detecção de códigos maliciosos. A implementação e o treinamento de um *autoencoder* torna-se um processo simples com o TensorFlow (ZACCONE *et al.*, 2017):

(i) As bibliotecas principais que vão ser utilizadas devem ser importadas.

(ii) O conjunto de dados de entrada deve ser carregado, comumente utilizando a função $input\_data$.

(iii) Os parâmetros da rede devem ser configurados, como a quantidade de exemplos e a taxa de aprendizado.

(iv) Uma variável do tipo *placeholder* deve ser definida para as entradas. O tipo de dados para esse sensor deve ser configurado como *float* e o formato como $[None, n\_input]$.

(v) Pesos e diferentes bias para a rede podem então ser definidos. A estrutura de pesos é para o codificador e o decodificador do *autoencoder*. São escolhidos utilizando o comando $tf.random\_normal$, que retorna valores aleatórios com uma distribuição normal.

(vi) A modelação da rede é dividida em duas partes complementares e conectadas: o codificador e decodificador. O codificador vai converter a entrada em informação útil para o *autoencoder*, além de comprimi-la. O decodificador vai descomprimir a entrada alterada para obter uma saída do mesmo tamanho que a entrada original.

(vii) Inicia a sessão e prepara um conjunto de amostras de treino para a rede, além de processos otimizadores existentes.

(viii) Finalmente, o modelo é testado aplicando o procedimento de codificação e decodificação para criar reconstruções que serão comparadas com os dados originais usando as capacidades do $matplotlib$ e $sklearn$.

O $matplotlib$ é uma biblioteca da linguagem de programação Python que é capaz de gerar gráficos partindo de dados contidos em listas e vetores no formato do $NumPy$, que é uma outra biblioteca no Python capaz de adicionar suporte para vetores multi-dimensionais e matrizes. O $NumPy$ fornece uma grande quantidade de métodos matemáticos pré-configurados e personalizáveis para serem utilizados pelos desenvolvedores. O $sklearn$ também é outra biblioteca livre, disponibilizando implementações utilizáveis dos algoritmos estudados na teoria do aprendizado de máquina.

O *autoencoder* que vai ser implementado através do TensorFlow e das bibliotecas mencionadas deve ser capaz de criar uma matriz de redução eficiente na reconstrução dos dados originais, minimizando a função de custo. Os resultados do *autoencoder* podem ser averiguados através de métodos contidos nas bibliotecas como o *classification_report* do $sklearn$. Além disso, o TensorFlow permite visualizar graficamente os resultados para comparação entre informações reconstruídas com as de entrada (ZACCONE *et al.*, 2017).

3 TRABALHOS RELACIONADOS

O problema da detecção de malware pode ser resumido da seguinte forma: em um conjunto de dados cada arquivo não-classificado deve receber uma classificação baseando-se no seu vetor de características criado pela arquitetura *Deep Learning*. Diversas abordagens foram utilizadas na área de Segurança Computacional para enfrentar o problema, como as que são citadas nesse capítulo: métodos de detecção por assinatura, mineração de dados, aprendizado de máquina, extrair as chamadas de API do Windows para classificação dos malwares e *Deep Learning*.

Métodos de detecção por assinatura são amplamente utilizados na industria anti-malware. Entretanto, autores de códigos maliciosos conseguem facilmente enganar esses métodos através de técnicas como criptografia (codificação de código para inviabilizar a sua análise), polimorfismo (constante alteração das formas do código sem modificar as funções do seu algoritmo) e ofuscação (tornar o código de difícil interpretação para sistemas detectores). Impulsionados pelos benefícios econômicos, a quantidade e diversidade de malwares aumentou nos anos recentes (HARDY *et al.*, 2016).

Novos sistemas para detecção de malwares foram desenvolvidos que aplicam mineração de dados e técnicas do aprendizado de máquina, com o propósito de fazer uma detecção automática e efetiva. Esses sistemas variam em seus usos de métodos para classificação e representação de características. Muitos propuseram associar os métodos de classificação as chamadas de API do Windows, através de análise após elas serem extraídas de arquivos executáveis (HARDY *et al.*, 2016).

Devido a sua habilidade em aprender atributos através de múltiplas camadas em uma arquitetura profunda, o *Deep Learning* permite aprender conceitos de alto nível baseando-se em representações locais de características. Como resultado, pesquisadores focaram suas atenções em métodos de *Deep Learning* em diversos domínios praticáveis. Nos anos recentes, esforços limitados de pesquisa foram devotados para a detecção de códigos maliciosos usando *Deep Learning*. Modelos desenvolvidos foram probabilísticos ou híbridos, utilizando ou não *autoencoders*. A seguir, uma lista com alguns modelos:

- Os autores de um modelo de relevância, chamado DeepAM, propuseram a utilização de *autoencoders* empilhados com máquinas restritas Boltzmann de múltiplas camadas e uma camada de memória associativa para conseguir detectar novos malwares desconhecidos. O modelo deles é uma estrutura *Deep*

Learning heterogênea para detecção inteligente de malware que realiza uma operação de treinamento gulosa com relação as camadas para aprendizado não-supervisionado, seguido por um ajusto fino supervisionado de parâmetros. Utiliza tanto arquivos de amostra classificados quanto não-classificados para pré-treinar múltiplas camadas do modelo com o propósito de aprendizagem de características (YE *et al.*, 2017).

• Uma outra metodologia de *Deep Learning* para classificação de malwares foi desenvolvida na Universidade de Hong Kong. Um modelo multi-tarefa para aprendizado que se baseia em sequência de chamadas da API para detectar e classificar códigos maliciosos. Utilizam redes neurais baseadas em *Autoencoders* para automatizar a representação de malwares detectados utilizando dimensões menores. Adicionalmente, implementam múltiplos decodificadores para obter mais informações além da classe ou família de um malware. Consiste de dois decodificadores: um classificador de malwares e o outro é gerador dos padrões para sequência das chamadas API nos acessos dos arquivos. O *framework* base é o seq2seq, que possui três configurações disponíveis para codificadores e decodificadores: um-para-muitos, muitos-para-um, muitos-para-muitos. O modelo utiliza um-para-muitos consistindo de: um codificador para aprendizado de representações, e dois decodificadores para classificar e gerar os padrões de acesso dos arquivos (WANG; YIU, 2016).

• MtNet é outro modelo de sistema *Deep Learning* multi-tarefa, desenvolvido na Universidade do Estado da Pensilvânia. Múltiplas camadas na arquitetura da Rede Neural são utilizadas para classificação de Malwares e o treino do sistema é feito com 4,5 milhões de arquivos. O sistema promove classificação binária(malicioso ou benigno) dos arquivos e também classifica famílias de malwares, sendo as duas tarefas combinadas na arquitetura multi-tarefa do sistema (HUANG; STOKES, 2016).

• Outro modelo que utiliza sequências de chamadas para classificação de Malwares foi um proposto por uma equipe de pesquisadores na Universidade Técnica de Munique. A rede neural combinada que eles construíram é baseada em camadas recorrentes e convolucionais com o objetivo de conseguir as melhores características para realizar a classificação, combinando convolução com modelação totalmente sequêncial (KOLOSNJAJI *et al.*, 2016).

• Para detectar malwares em sistema Android, como o visto na Figura 12,

foram feitos os métodos HADM e o DroidDetector. O primeiro é uma análise híbrida para detecção de malwares que extraí informações estáticas e dinâmicas para converte-las em representações vetoriais com a utilização de uma Rede Neural Profunda (DNN), e também com auxílio de kernels diversos para converter a informação dinâmica em representações em grafo que terão seus resultados de aprendizado combinados com as vetoriais para construir um modelo de classificação híbrido (XU *et al.*, 2017). Já o DroidDetector é uma engine para um sistema *Deep Learning* online de detecção de malwares que automaticamente identifica se um aplicativo é malware ou não. Os tipos de características que o DroidDetector extraí para detectar um malware são: permissões requisitadas, APIs de risco e comportamentos dinâmicos. As duas primeiras são extraídas através de análise estática, enquanto a última é extraída através de análise dinâmica. O DroidDetector somente precisa do arquivo .apk de instalação de cada aplicativo Android para realizar a verificação (YUAN *et al.*, 2016).

- Na quadragésima conferência anual de software e aplicativos para computar da IEEE foi proposto um método que foca em medidas para combater uma infecção de um malware. Este método aborda a detecção de malwares utilizando o comportamento de processos em terminais possivelmente infectados, observando o tráfico de dados que vem de um malware. O modelo treina uma rede neural recorrente para extrair características do comportamento de um processo, depois treina um rede neural convolucional para classificar representações das características que foram geradas pela rede recorrente treinada (TOBIYAMA *et al.*, 2016).

- Por fim, há um sistema Deep Learning conceptualizado em algoritmos geradores de domínio, *Domain Generation Algorithm*(DGA) em inglês, chamado de DeepDGA cujo propósito é intencionalmente enganar um detector de malwares baseado em Deep Learning. O objetivo é treinar o modelo de detecção para compensar a estratégia do adversário de gerar nomes de domínios mais difíceis de detectar. A arquitetura deles evolui com a utilização de amostras adversárias para dar ao classificador aleatório do sistema maior eficiência em detectar famílias de malware que utilizam DGAs e que não foram conhecidos pelo sistemas durante a fase de treinamento (ANDERSON *et al.*, 2016).

4 MODELO PROPOSTO

A arquitetura que será implementada é de *Deep Learning* utilizando um (AE), para treinamento não supervisionado de características e ajuste supervisionado de parâmetros(pesos e vetores de vieses). O trabalho utiliza o modelo das AEs para aprender características genéricas de um malware, sendo capaz de auxiliar classificadores em detectar códigos maliciosos novos que não sejam conhecidos.

Os *Autoenconders* são autoassociadores. Cada *AutoEnconder* é um bloco da rede neural profunda que é composto de duas partes: um *Encoder* que reduz a dimensionalidade da entrada e um Decoder que busca retornar a saída do *Encoder* para o estado original. A saída do *Encoder* será usada como entrada para a próxima camada ao qual se conecta, ou seja, o Decoder. O sistema modelado consistirá de três componentes principais: o extraidor de características, a rede neural do tipo Auto-Encoder e o classificador. Não haverá a presença de múltiplos *AutoEnconders* neste trabalho devido a limitações técnicas com tempo de treino dos *AutoEnconders*, quantidade de amostras de malwares e poder de processamento dos dispositivos utilizados para o desenvolvimento do sistema detector de códigos maliciosos.

O construtor de características será composto de um Descompressor e um *Parser* das *Portable Executable*(PE) do Windows. Uma PE é um formato de arquivos usado pelo sistema operacional Windows. O Descompressor é utilizado caso o arquivo PE esteja comprimido. O *Parser* é um software de análise sintática que captura uma entrada e constrói uma estrutura de dados para representá-la. Também verifica se a sintaxe está correta. Os arquivos PEs são convertidos pelo *Parser* para um conjunto de IDs globais de 32-bit que representam suas funções e podem ser armazenadas no banco de assinaturas. O *Parser* utilizado no modelo proposto vai ser um programa que seja capaz de extrair de cada executável a sua sequência de chamadas das funções na API do Windows.

A primeira camada da rede neural vai receber um conjunto de dados para treino como sua entrada. As camadas subsequentes vão receber as saídas das camadas do tipo *enconder* e *decoder* até a última, que vai ser a saída do AutoEncoder ao todo. O objetivo do AutoEncoder padrão é reconstruir as amostras de dados que recebe na sua saída, aprendendo a criar representações dimensionalmente menores nas suas camadas escondidas.

A última parte do modelo vai ser o classificador, utilizado para retornar na saída os relatórios de detecção de malware. O classificador aplicado no modelo pro-

posto não é apenas um, mas vários que são abordados na teoria do aprendizado de máquina. Alguns dos classificadores da literatura que foram testados: Árvores de Decisão, Naive Bayes Gaussiano e Regressão Logística. Cada classificador foi implementado individualmente no modelo e cada resultado obtido foi coletado.

Os classificadores utilizados tem o propósito de validar a eficiência do *Autoencoder* que foi desenvolvido em TensorFlow, treinado e ajustado. A parte de treino e ajuste é da arquitetura *Deep Learning*: realizando aprendizado não-supervisionado de características, ajuste fino supervisionado e a detecção indireta dos malwares através de uma técnica especial de treino que vai ensinar o Autoencoder a reconstruir arquivos de modo que malwares sejam priorizados e executáveis benignos sejam ignorados. Esse método especial de treino para o Autoencoder é detalhado no Capítulo 5 que aborda os resultados deste trabalho. O sistema projetado é ilustrado pela Figura 18.

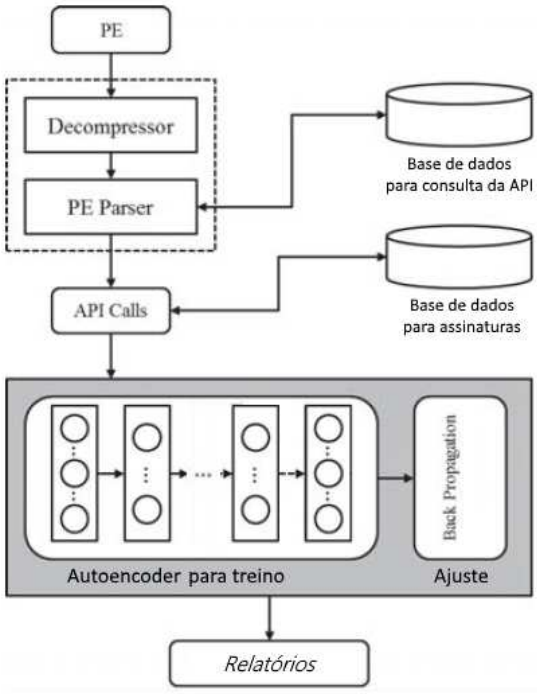

Figura 18 – Arquitetura do Sistema *Deep Learning* que será modelado
Fonte: Adaptado de Hardy (2016, p. 62).

O comportamento em pedaços de código dos arquivos podem ser organizados em vetores através da análise de chamadas da API do Windows, como mostra a tabela 1. Um arquivo f_i em um conjunto de dados D pode ser denotado na forma de (A_{f_i}, V_{f_i}), em que A_{f_i} é o vetor de características do arquivo baseado nas chamadas da API do Windows e V_{f_i} é a classe do arquivo. M_a são arquivos benignos, M_b são indeterminados e M_c são os maliciosos.

Tabela 1 – Exemplo de um conjunto de dados para arquivos amostrais

Arquivo	API *Calls* extraídas	Vetor de Características	Classe de Malware
$f1$	API_1, API_3, API_5	[1, 0 , 1, 0 , 1, 0]	M_a
$f2$	API_1, API_2, API_3, API_5	[1, 1 , 1, 0 , 1, 0]	M_a
$f3$	API_1, API_6	[1, 0 , 0, 0 , 0, 1]	M_b
$f4$	API_2, API_3	[0, 1 , 1, 0 , 0, 0]	M_c
$f5$	API_1, API_2, API_3, API_4	[1, 1 , 1, 1 , 0, 0]	M_c
$f6$	API_5	[0, 0 , 0, 0 , 1, 0]	M_b

Fonte: Autoria própria.

Seja n o número das chamadas de API do Windows que foram extraídas do conjunto de dados D. O vetor de características para cada arquivo pode ser representado por:

$$A_{f_i} = (a_{i1}, a_{i2}, ..., a_{in}) \tag{43}$$

No qual a_{ij} pode ser de valor 1, quando contém uma chamada de API_x, ou 0 se esse não for o caso.

Formalmente, a estrutura do *Autoencoder* que é utilizado pode ser definida como a seguir: uma parte é o codificador para transformar um vetor de entrada x_i em uma representação vetorial escondida y_i através de um mapeamento determinístico f. A outra parte é o decodificador que é capaz de mapear a representação escondida y_i para uma reconstrução z_i com mesma dimensão da entrada x_i através de uma função g. f e g podem ser descritos pelas equações abaixo:

$$y_i = f(x_i) = s(W_{x_i} + B) \tag{44}$$

$$z_i = g(x_i) = s(w_{y_i} + b) \tag{45}$$

Sendo que W é uma matriz de pesos, B é um vetor de *offsets* (ou vieses), e w e b são transformadas com dimensões menores que as da entrada e saída, respectivamente. w e b vão ter a mesma dimensão das camadas escondidas.

Tipicamente, o número de camadas escondidas será menor que a quantidade de camadas visíveis para forçar o *Autoencoder* a realizar uma redução da dimensionalidade dos dados de entrada, comprimindo o vetor em representações menores e

tentar reconstruí-lo mantendo apenas características relevantes. Por fim, há o s que é uma função sigmoide denominada por:

$$s(t) = \frac{1}{1 + exp^{-t}} \tag{46}$$

O processo de treinamento deve minimizar o erro na reconstrução, encontrando a melhor representação comprimida para dados de entrada. O algoritmo a seguir ilustra o procedimento geral de treinamento em um *autoencoder*:

Entrada: Conjunto de Dados X com n amostras de treinamento: X_i = (A_{f_i}, V_{f_i}), onde i vai de 1 até n e V_{f_i} tem valores dos tipos M_a, M_b e M_c;
Saída: Configuração de Parâmetros W e B
Inicializar(W,B);
enquanto *o erro de treino* E *não convergir ou a iteração desejada não for alcançada* **faça**
  para *cada entrada* X_i **faça**
    Computar ativações y_i na camada escondida, obter a saída z_i na camada de saída;
  fim
  Calcular o erro de treino $E(x, z)$;
  Utilizar *backpropagation* para propagar o erro para trás pela rede e atualizar o conjunto de parâmetros W e B;
fim

Observa-se que quaisquer outras funções de perda e parametrização podem ser implementadas nos codificadores e decodificares além das que são padrão da literatura.

O treinamento de uma rede neural pode ser demasiado complexo na medida em que a escala da rede aumenta, podendo exigir tanto um poder de processamento computacional proibitivo para pesquisadores sem financiamento quanto um tempo de espera inviável para o término do treino. Levando em conta as limitações de tempo e dispositivos disponíveis pelo autor deste trabalho, somente será desenvolvido um *autoencoder* com quatro camadas escondidas na sua composição: duas para o *encoder* e as outras para o *decoder*.

5 RESULTADOS

5.1 ESPECIFICAÇÕES TÉCNICAS

O computador utilizado para este trabalho foi um *notebook* com 16 GB de memória RAM, um processador Core i7-6700HQ e uma placa de Vídeo NVIDIA GeForce GTX 960M. Uma Máquina Virtual do Windows 10 foi montada via *software* da Oracle chamado VirtualBox (versão 5.2.18 r124319) para tratar da extração de malwares e os seus atributos desejados.

O ambiente de desenvolvimento do trabalho foi em um Jupyter Notebook, que é uma aplicação virtual de código aberto que permite a criação, gerenciamento e compartilhamento de documentos que contenham equações, imagens, texto e código de programação executável em células. Foi utilizado para limpar as bases de dados, transformar os dados em valores numéricos, visualizar as tabelas de dados, construir os métodos do aprendizado de máquina que foram utilizados e entre outras necessidades do trabalho.

O software utilizado para extrair as sequências de chamadas das funções na API do Windows foi o API Monitor. A base de dados dos malwares foi derivada de uma base pública que é mantida por um pesquisador e chamada de APIMDS (KIM, 2018). A base de dados benignos foi extraída manualmente pelo autor deste trabalho com auxílio do programa extrator de *API Calls*. A taxa de aprendizado do *Autoencoder* foi de 0.00001 e ele tinha apenas quatro camadas: duas para o *Encoder* e outras duas para o seu *Decoder*.

5.2 *AUTOENCODER*

O método utilizado neste trabalho treina o *Autoencoder* para que reconstrua na sua saída uma forma modificada da entrada. O método é simples: o *Autoencoder* vai receber na entrada a base de dados original, mas na saída ele vai ter como alvo somente os dados originados de malwares enquanto os dados provindos de arquivos benignos terão seus valores zerados. Neste caso, o *Autoencoder* não estará só reduzindo a dimensionalidade da entrada nas camadas escondidas: também vai aprender no *Encoder* como mapear as entradas para um subgrupo cuja estrutura matemática é diferente da entrada, ou seja, não vai apenas ser uma compressão da entrada.

Para gerar a base de treino, os dados extraídos dos programas coletados para este trabalho foram pré-processados com um codificador de marcações. As classes

de atributos foram normalizadas em valores que foram recebidos pelos classificadores para realizar as predições.

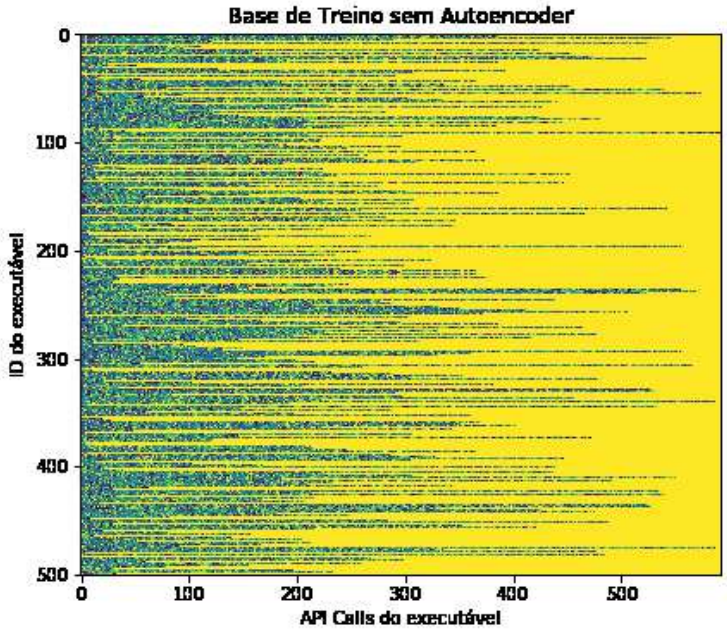

Figura 19 – **Espectrograma da base original, pré-processada das listas de sequências de *API Calls* dos binários.**

Fonte: Autoria própria.

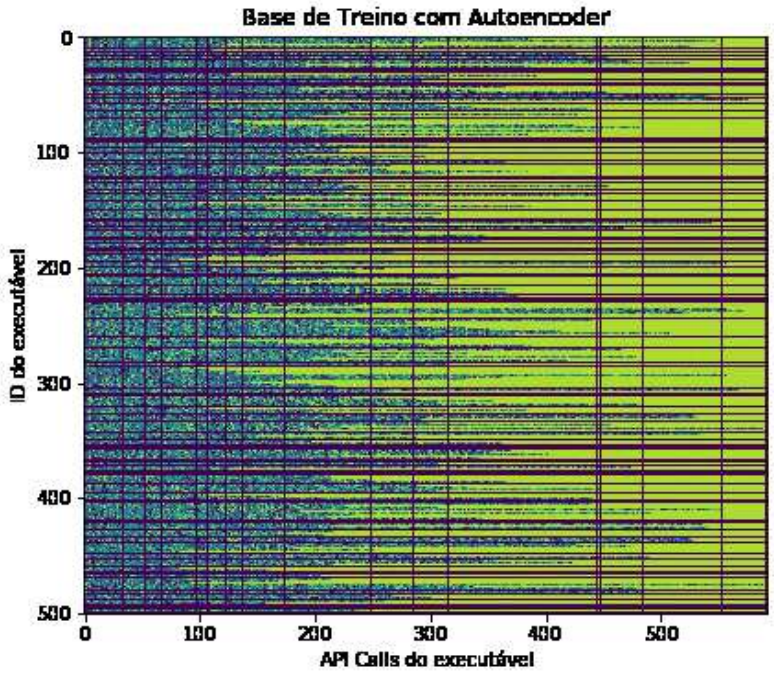

Figura 20 – **Espectrograma da base gerada pelo Autoencoder após processar a base original.**
Fonte: Autoria própria.

As Figuras 19 e 20 ilustram os espectrogramas da base de dados, respectivamente antes e depois de serem processados pelo *Autoencoder*. O eixo X dos espectrogramas, chamado de *API Calls* do executável, representa a posição de uma

API Call na sequência de chamadas feitas por um executável. O eixo Y é o ID do executável e representa o índice de cada executável, ou seja, a qual executável vai pertencer uma determinada sequência de *API Calls*. Os diferentes tons nos espectrogramas representam os valores atribuídos para cada *API Call* pelo codificador de rótulos e o normalizador.

Observe no espectrograma da base original que certos arquivos terão sequências de *API Calls* mais longas que outros. Isso ocorre em grande parte por causa dos malwares na base de dados que é utilizada para treino e teste, pois suas sequências podem chegar ao comprimento de até 598 *API Calls* enquanto os arquivos benignos podem alcançar só até 453 *API Calls* no máximo. Além disso, em média os malwares vão possuir mais *API Calls* que os arquivos benignos devido a necessidade deles de realizarem funções além daquelas realizadas por programas comuns para conseguirem praticar atividades ilícitas nos seus sistemas hospedeiros.

Observe também que o espectrograma da base de treino original (sem *Autoencoder*) difere do espectrograma da base que o *Autoencoder* gerou depois de realizar o processamento da base original nas suas camadas escondidas, mesmo que tenham algumas características parecidas. Em outros casos, é frequente que a saída de um *Autoencoder* seja parecida ou até completamente a mesma coisa que a entrada por causa do treinamento padrão dos *Autoencoders* que é utilizar a própria entrada como alvo da saída. Porém, neste trabalho isso não vai ocorrer porque deseja-se que a saída não seja uma cópia da entrada e sim uma versão realçada que apenas foque nos dados de malwares, ou seja, que o alvo da saída zere os valores de *API Calls* que não sejam dos malwares durante o treinamento do *Autoencoder*. Como é desejado que arquivos benignos sejam ignorados, então é esperado que a saída fique diferente da entrada e também é esse o objetivo.

5.3 *CLASSIFICADORES*

Por razão da quantidade de amostras ser restrita, o treinamento de nenhum dos classificadores ultrapassou mais que 10 minutos. O tempo de treinamento foi pequeno para todos (em comparação ao AutoEncoder, que demorou quase dois dias) ocorre devido ao número pequeno de amostras utilizadas para treinamento dos classificadores (300 arquivos benignos e 700 malwares). Os classificadores utilizados foram modelos clássicos da literatura, implementados através de uma biblioteca livre de algoritmos para aprendizado de máquina que é chamada de scikit-learn(ou Sklearn).

Possui diversos métodos para classificação como: *Support Vector Machine*(SVM), *k-nearest neighbors*(KNN), e muitos outros classificadores que não entraram no contexto deste trabalho como *Random Forests* e *Gradient Tree Boosting*.

Depois de treinados, os classificadores receberam uma porção da base original para treinamento e outra parte dessa base de dados para validar as suas previsões. Através do treinamento os classificadores aprenderam diversas regras de previsão, baseadas nos parâmetros que extraem dos dados recebidos para realizar previsões. Não está dentro do escopo atual explicar as regras de cada classificador, mas em termos leigos: a Árvore de Decisão cria uma tabela de condições em formato de uma árvore que vai processar os dados até serem classificados pelos últimos ramos, o Naive Bayes Gaussiano vai utilizar uma abordagem probabilística que assume cada atributo de entrada como uma variável independente das outras no cálculo da probabilidade de uma amostra de dados corresponder a uma determinada classe, o KNN vai avaliar a proximidade entre elementos conhecidos com seus k vizinhos para encaixar cada ponto de dados em suas possíveis classes, a SVM tenta encontrar a lacuna mais ampla entre os pontos conhecidos que melhor separe as amostras de classes diferentes no espaço dos dados, a Regressão Logística é outro modelo matemático utilizado em Estatística para prever a probabilidade de um evento baseando-se em dados anteriores, e o *Multilayer Perceptron* é somente um *Perceptron* com várias camadas escondidas (apenas uma camada para o caso deste trabalho). Também é preciso explicar a métrica de avaliação utilizada neste trabalho: o F1-Score. O F1-Score é a média harmônica entre a Precisão e o *Recall* de cada classificador.

A Precisão é entendida como a proporção de previsões positivas reais feitas pelo classificador, em comparação ao conjunto de previsões positivas feitas. Ou seja, a Precisão é o percentual de vezes que uma determinada classe foi corretamente prevista pelo classificador. Já o *Recall* é entendido como a proporção de previsões positivas obtidas pelo classificador, em comparação ao número real de amostras positivas existentes. Ou seja, o Recall é um percentual de obtenção que o classificador possui para uma determinada classe. Abaixo estão as fórmulas que explicam cada métrica, incluindo a F1-Score que foi utilizada como marco para avaliar os resultados de cada classificador.

$$Precisão = \frac{Verdadeiros\ Positivos}{Verdadeiros\ Positivos + Falsos\ Positivos} \tag{47}$$

$$Recall = \frac{Verdadeiros\ Positivos}{Verdadeiros\ Positivos + Falsos\ Negativos} \tag{48}$$

$$F1\text{-}Score = \frac{2 * (Recall * Precisão)}{Recall + Precisão} \tag{49}$$

Na Tabela 2, são observados os resultados obtidos no experimento após o treinamento dos classificadores. Para a base original, todos obtiveram resultados acima de 80% com exceção do Naive Bayes Gaussiano. Isso pode ter ocorrido pela natureza dos malwares e arquivos benignos na base de dados, sendo os malwares analisados possivelmente fracos em comparação aos mais perigosos que causam prejuízos para grandes empresas todo ano. Também é possível que utilizar a extração de *API Calls* seja apenas mais eficiente do que o esperado, embora análises com bases de dados maiores seriam necessárias para validar isso.

Tabela 2 – Resultados obtidos com o teste dos classificadores

Tipo de Classificador	Resultados (F1-Score)			
	Base Original		Otimização com *Encoder*	
	Benigno	Malware	Benigno	Malware
Árvore de Decisão	91%	96%	91%	97%
Naive Bayes Gaussiano	68%	76%	91%	96%
KNN	90%	96%	91%	97%
SVM	84%	92%	91%	96%
Regressão Logística	92%	96%	91%	97%
Multilayer Perceptron	90%	96%	91%	96%

Fonte: Autoria própria.

Em seguida, na Tabela 2 estão os resultados obtidos para a base após ela ser otimizada com o *Encoder* do *Autoencoder*. Para este trabalho, otimizar significa que tiveram seu tamanho reduzido pelo *Encoder* e suas informações processadas para realçar os dados de malwares que foram observados pelo treinamento do *Autoencoder*.

Cada amostra original possui 593 variáveis, que foram reduzidas para um tamanho de 197 variáveis pelo *Encoder*. Portanto, a redução do tamanho de cada amostra foi em torno de 67% após serem processadas pelo *Encoder*.

A otimização revelou um resultado que atendeu as expectativas deste trabalho: todos os classificadores melhoraram na sua classificação, visto que mantiveram patamares superiores de F1-Score para a detecção dos malwares mesmo utilizando uma base da dados menor que a original. Ou seja, isso não só mostra que o *Auto-*

encoder produziu no seu *Encoder* uma representação menos custosa para os classificadores em termos de processamento dos dados como também conseguiu manter níveis de informação dentro dessas representações que são superiores aos dados originais.

O Notebook Jupyter com todos os códigos dos classificadores e do Autoencoder utilizado está disponível no Github do autor deste trabalho[1].

[1] **https://github.com/leocsato**

6 CONCLUSÃO

O *Autoencoder* desenvolvido neste trabalho é útil para qualquer método classificador, visto que retorna uma representação mais informativa e menor das *API Calls* extraídas de executáveis. Não só seria útil para diminuir o custo de processamento em bases muito maiores de dados, como também pode ajudar bases menores a identificar malwares de forma mais eficiente. O método desenvolvido neste trabalho de criar um *Autoencoder* que zera dados indesejados foi parcialmente inspirado pelos *Autoencoders* removedores de ruídos, mas seguindo o conceito oposto de introduzir ruído em pontos estratégicos da base de dados. Não é uma ideia popular na área dos *Autoencoders*, mas fica evidente neste trabalho a sua utilidade nos cenários adequados.

O objetivo geral deste trabalho foi desenvolver uma rede neural profunda que otimizasse a detecção de malwares, o que foi cumprido pelo *Autoencoder* criado. Os objetivos específicos também foram concluídos ao decorrer do desenvolvimento deste trabalho, sendo que o último objetivo teve suas expectativas superadas embora seria preciso uma validação dos resultados obtidos com uma base da dados mais extensa. Visto a natureza dos malwares, cujo propósito é enganar pessoas e computadores para realizarem atividades ilícitas, é preciso maiores margens de segurança antes que um detector de malwares seja afirmado como capaz de detectar mais que 60% de todos os malwares. Mesmo assim, os resultados obtidos foram promissores e sugerem que a possibilidade de uma investigação mais aprofundada do método utilizado no *Autoencoder* deveria ser considerada, assim também como a possibilidade de melhorar os resultados obtidos abordando outros aspectos do trabalho desenvolvido.

6.1 DESENVOLVIMENTOS FUTUROS

Um possível rumo para aprimorar este trabalho seria otimizar o processo de treinamento utilizando placas gráficas dedicadas ou outra biblioteca além do TensorFlow. O desenvolvimento de uma ferramenta própria para a extração de características também poderia ser considerado. Outra opção adicional seria desenvolver um sistema de processamento sequencial para cada executável, detectando em tempo real a presença de malwares. Além disso, é interessante investigar a utilização de classificadores baseados em arquiteturas mais complexas de redes neurais que as vistas neste trabalho. Alguns exemplos são as Redes Neurais Convolucionais e as Recorrentes.

REFERÊNCIAS

ANDERSON, Hyrum S.; WOODBRIDGE, Jonathan; FILAR, Bobby. DeepDGA: Adversarially-tuned domain generation and detection. In: **AISec '16: Proceedings of the 2016 ACM Workshop on Artificial Intelligence and Security**. New York, NY: ACM, 2016.

BASTOS, Ian Vilar (Ed.). **XVI Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais**. Niterói, RJ: Sociedade Brasileira de Computação, 2016.

BERKELEY AI RESEARCH. **Caffe: a fast open framework for deep learning.** 2014. https://github.com/BVLC/caffe. Acesso em: 26 de nov. 2018.

BHATTACHARYYA, Dhruba Kumar; KALITA, Jugal Kumar. **Network Anomaly Detection: A Machine Learning Perspective**. Boca Raton, FL: CRC Press, 2014.

BLANCHETTE, Jasmin. **The Little Manual of API Design**. Junho 2008. https://people.mpi-inf.mpg.de/~jblanche/api-design.pdf. Acesso em: 09 de set. 2017.

BUGHIN, James; HAZAN, Eric; RAMASWAMY, Sree; CHUI, Michael; ALLAS, Tera; DAHLSTROM, Peter; HENKE, Nicolaus; TRENCH, Monica. **Artificial Intelligence: The Next Digital Frontier?** Junho 2017. https://www.mckinsey.com/business-functions/mckinsey-analytics/our-insights/how-artificial-intelligence-can-deliver-real-value-to-companies. Acesso em: 09 de set. 2017.

COMODO GROUP. **Comodo Security Solutions, Inc**. 2018. https://antivirus.comodo.com/. Acesso em: 26 de nov. 2018.

CUTLER, Silas. **The MalShare Project**. 2018. http://www.malshare.com/. Acesso em: 26 de nov. 2018.

CYLANCE, Data Science Team. **Introduction to Artificial Intelligence for Security Professionals**. IRVINE, CA: THE CYLANCE PRESS, 2017.

DAMIEN, Aymeric. **TFLearn: Deep learning library featuring a higher-level API for TensorFlow.** 2016. https://github.com/tflearn. Acesso em: 26 de nov. 2018.

DAUMÉ III, Hal. **A Course in Machine Learning**. Janeiro 2017. http://ciml.info/dl/v0_99/ciml-v0_99-all.pdf. Acesso em: 09 de set. 2017.

DENG, Li; YU, Dong. **Deep Learning: Methods and Applications**. Hanover, MA: Now Publishers Inc., 2014.

DUA, Sumeet; DU, Xian. **Data Mining and Machine Learning in Cybersecurity**. Boca Raton, FL: CRC Press, 2011.

EASTTOM, Chuck. **Computer Security Fundamentals**. Second. Indianapolis, Indiana: Pearson, 2012.

FORENSICATION. **VirusShare.com: Because Sharing is Caring**. 2018. https://virusshare.com/. Acesso em: 26 de nov. 2018.

GOODFELLOW, Ian; BENGIO, Yoshua; COURVILLE, Aaron. **Deep Learning**. Cambridge, MA: MIT Press, 2016. (Adaptive computation and machine learning series). http://www.deeplearningbook.org. Acesso em: 09 de set. 2017.

GUPTA, Sanchit; KAUR, Sarvjeet; SHARMA, Harshit. **Malware Characterization using Windows API Call Sequences**. 2016. http://www-users.math.umn.edu/~math-sa-sara0050/space16/slides/space2016121708-06.pdf. Acesso em: 09 de set. 2017.

GUYON, Isabelle; ELISSEEFF, André. **Feature Extraction: Foundations and Applications**. Berlin, Heidelberg: Springer, 2006.

HARDY, William; CHEN, Lingwei; HOU, Shifu; YE, Yanfang; LI, Xin. Dl4md: A deep learning framework for intelligent malware detection. **The 12th International Conference on Data Mining 2016(DMIN'16)**, CSREA Press, v. 12, 2016.

HAYKIN, Simon. **Redes Neurais: Princípios e prática**. Porto Alegre: Bookman, 2008.

HAYKIN, Simon. **Neural Networks and Learning Machines**. Upper Saddle River, New Jersey: Pearson Prentice Hall, 2009.

HUANG, Wenyi; STOKES, Jack W. MtNet: A multi-task neural network for dynamic malware classication. In: **Detection of Intrusions and Malware, and Vulnerability Assessment: 13th International Conference, DIMVA 2016**. New York, NY: Springer-Verlag New York, 2016. (Security and Cryptology, v. 9721), p. 399–418.

IBM. **IBM Watson APIs: A collection of REST APIs and SDKs that use cognitive computing to solve complex problems.** 2018. https://github.com/watson-developer-cloud. Acesso em: 26 de nov. 2018.

JACOBS, Jay; RUDIS, Bob. **Data Driven Security: Analysis, Visualization and Dashboards**. Indianapolis, Indiana: Wiley, 2014.

JOLIBRAIN. **DeepDetect: Deep Learning API and Server in C++11 support for Caffe, Caffe2, Dlib, Tensorflow, XGBoost and TSNE.** https://github.com/jolibrain/deepdetect. Acesso em: 26 de nov. 2018.

JUNG, Wookhyun; KIM, Sangwon; CHOI, Sangyong. Deep learning for zero-day flash malware detection. **36th IEEE Symposium on Security and Privacy**, IEEE, San Jose, v. 36, 2015.

KIM, Huy Kang. **APIMDS (API-based malware detection system)**. 2018. http://ocslab.hksecurity.net/apimds-dataset. Acesso em: 01 de ago. 2018.

KOLOSNJAJI, Bojan; ZARRAS, Apostolis; WEBSTER, George; ECKERT, Claudia. Deep learning for classication of malware system call sequences. In: **AI 2016: Advances in Artificial Intelligence**. [S.l.]: Springer Cham, 2016. (Lecture Notes in Computer Science, v. 9992).

LI, Yuancheng; MA, Rong; JIAO, Runhai. A hybrid malicious code detection method based on deep learning. **International Journal of Security and Its Applications**, SERSC, v. 9, n. 5, p. 205–216, 2015.

MALWAREBYTES. **State of Malware Report 2017**. Santa Clara, CA, 2017.

MAZUR, Matt. **A Step by Step Backpropagation Example**. 2015. https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/. Acesso em: 01 de nov. 2017.

PONEMON INSTITUTE. **2016 Cost of Data Breach Study: Global Analysis**. Traverse City, Michigan, Junho 2016.

POTIER, Jacquelin. **WinAPIOverride : Free Advanced API Monitor**. 2018. http://jacquelin.potier.free.fr/winapioverride32/. Acesso em: 10 de nov. 2018.

RANA, Anurag; EISENSON, Andrew; NOSELLI, Caitlin; BASON, Tamlin; MUNTHE-KAAS, Erlend; CHRISTOU, Edmond. **Cyberattacks and the threat to the global economy**. Julho 2017. https://www.bloomberg.com/professional/blog/cyberattacks-threat-global-economy/. Acesso em: 01 de nov. 2018.

SCOTT, James. **Signature Based Malware Detection is Dead**. Washington D.C.: Institute for Critical Infrastructure Technology, 2017.

SIKORSKI, Michael; HONIG, Andrew. **Practical Malware Analysis**. San Francisco, CA: No Starch Press, 2012.

SOUPPAYA, Murugiah; SCARFONE, Karen. **Guide to Malware Incident Prevention and Handling for Desktops and Laptops**. Gaithersburg, MD: National Institute of Standards and Technology, 2013.

SYMANTEC CORPORATION. **A-Z Listing of Threats & Risks**. 2018. https://www.symantec.com/security-center/a-z. Acesso em: 26 de nov. 2018.

TOBIYAMA, Shun; YAMAGUCHI, Yukiko; SHIMADA, Hajime; IKUSE, Tomonori; YAGI, Takeshi. Malware detection with deep neural network using process behavior. In: **2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)**. [S.l.]: IEEE, 2016. v. 2, p. 577–582.

VEEN, Fjodor van. **A mostly complete chart of Neural Networks**. 2016. https://www.asimovinstitute.org. Acesso em: 01 de nov. 2017.

WANG, Xin; YIU, Siu Ming. A multi-task learning model for malware classification with useful file access pattern from api call sequence. **ArXiv CoRR**, abs/1610.05945, Outubro 2016.

WUEEST, Candid. **Financial Threats Review 2017: An ISTR Special Report. Internet Security Threat Report(ISTR)**. Mountain view: Symantec Corporation, 2017.

XU, Lifan; ZHANG, Dongping; JAYASENA, Nuwan; CAVAZOS, John. Hadm: Hybrid analysis for detection of malware. In: **Proceedings of SAI Intelligent Systems Conference (IntelliSys) 2016**. [S.l.]: Springer Cham, 2017. (Lecture Notes in Networks and Systems, v. 16).

YE, Yanfang; CHEN, Lingwei; HOU, Shifu; HARDY, William; LI, Xin. Deepam: a heterogeneous deep learning framework for intelligent malware detection. **Knowledge and Information Systems**, p. 1–21, Abril 2017.

YUAN, Zhenlong; LU, Yongqiang; XUE, Yibo. Droiddetector: Android malware characterization and detection using deep learning. **Tsinghua Science and Technology**, v. 21, n. 1, p. 114–123, Fevereiro 2016.

ZACCONE, Giancarlo; KARIM, Md.Rezaul; MENSHAWY, Ahmed. **Deep Learning with TensorFlow**. Birmingham, West Midlands: Packt Publishing Ltd., 2017.

ZADEH, Reza; RAMSUNDAR, Bharath. **TensorFlow for Deep Learning**. Sebastopol, CA: O'Reilly Media, 2017.