SISTEMA DE DETECÇÃO DE ANOMALIAS EM SISTEMAS EMBARCADOS

Trabalho de Conclusão de Curso

Engenharia de Computação

Tiago Farias Cabral
Orientador: Prof. M.Sc. José Paulo Gonçalves de Oliveira

UNIVERSIDADE DE PERNAMBUCO

Tiago Farias Cabral

SISTEMA DE DETECÇÃO DE ANOMALIAS EM SISTEMAS EMBARCADOS

Monografia apresentada como requisito parcial para obtenção do diploma de Bacharel em Engenharia de Computação pela Escola Politécnica de Pernambuco – Universidade de Pernambuco.

UNIVERSIDADE DE PERNAMBUCO
ESCOLA POLITÉCNICA DE PERNAMBUCO
GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

Recife, março de 2021

Cabral, Tiago Farias
Sistema de detecção de anomalias em sistemas embarcados. / Tiago Farias Cabral – Recife - PE, 2021.
xiv, 37 f. : il. (algums color.) ; 29cm.

Trabalho de Conclusão de Curso (Graduação em Engenharia de Computação) Universidade de Pernambuco, Escola Politécnica de Pernambuco, Recife, 2021.

Orientador: Prof. M.Sc. José Paulo Gonçalves de Oliveira.

Inclui referências.

1. Sistemas embarcados. 2. Detecção de anomalias. I. Título. II. Orientador G. de Oliveira, José Paulo. III. Universidade de Pernambuco.

MONOGRAFIA DE FINAL DE CURSO

Avaliação Final (para o presidente da banca)*

No dia 11/5/2021, às 09h30min, reuniu-se para deliberar sobre a defesa da monografia de conclusão de curso do(a) discente **TIAGO FARIAS CABRAL**, orientado(a) pelo(a) professor(a) **JOSÉ PAULO G. DE OLIVEIRA**, sob título Sistema De Detecção De Anomalias Em Sistemas Embarcados, a banca composta pelos professores:

BRUNO JOSÉ TORRES FERNANDES (PRESIDENTE)
JOSÉ PAULO G. DE OLIVEIRA (ORIENTADOR)

Após a apresentação da monografia e discussão entre os membros da Banca, a mesma foi considerada:

☒ Aprovada ☐ Aprovada com Restrições* ☐ Reprovada

e foi-lhe atribuída nota: 10,0 (DEZ)

*(Obrigatório o preenchimento do campo abaixo com comentários para o autor)

O(A) discente terá 7 dias para entrega da versão final da monografia a contar da data deste documento.

AVALIADOR 1: Prof (a) **BRUNO JOSÉ TORRES FERNANDES**

AVALIADOR 2: Prof (a) **JOSÉ PAULO G. DE OLIVEIRA**

AVALIADOR 3: Prof (a)

* Este documento deverá ser encadernado juntamente com a monografia em versão final.

Agradecimentos

Agradeço primeiramente a Deus e aos meus pais, Maurício e Valéria Cabral, que sempre batalharam para proporcionar boa educação e boas condições para minha família. Também aos meus irmãos, Rafael e Amanda, por acreditarem no meu potencial e me acompanharem em todos os momentos, me guiando por caminhos de desafios e sucesso.

Aos meus amigos, que sempre me aconselharam e encorajaram a seguir em frente e buscar o meu melhor, mesmo em momentos difíceis, minha eterna gratidão.

Um agradecimento especial ao meu professor orientador, José Paulo Gonçalves de Oliveira, pela grande atenção e apoio prestados.

Por fim, agradeço a Deus por estar sempre presente em minha vida, me fazendo enxergar que o melhor está por vir e me dando força e perseverança para acreditar que sou capaz.

ESCOLA POLITÉCNICA
DE PERNAMBUCO

Autorização de publicação de PFC

Eu, **Tiago Farias Cabral** autor(a) do projeto de final de curso intitulado: **Sistema De Detecção De Anomalias Em Sistemas Embarcados;** autorizo a publicação de seu conteúdo na internet nos portais da Escola Politécnica de Pernambuco e Universidade de Pernambuco.

O conteúdo do projeto de final de curso é de responsabilidade do autor.

Tiago Farias Cabral
_____
Tiago Farias Cabral

_____
Orientador(a): **José Paulo G. de Oliveira**

_____
Coorientador(a):

_____ _____
Prof, de TCC: **Daniel Augusto Ribeiro Chaves** Data: 11/5/2021

Resumo

A constante necessidade de otimizar processos nas indústrias fomentou o uso de sistemas embarcados como solução para problemas dos mais diversos segmentos. Essa tendência tem tornado a complexidade das placas de circuito impresso (PCI) cada vez maior. Em contrapartida, essa maior complexidade gera um aumento significativo na probabilidade de falhas para os sistemas compactos. O problema é que a habilidade humana e os métodos tradicionais de testagem se tornaram insuficientes para identificar a presença de falhas no sistema. Tendo em vista essa dificuldade de identificação, este trabalho propõe a implementação em plataforma compacta de um sistema não-invasivo de detecção de anomalias, permitindo a verificação de sistemas embarcados e a classificação do seu funcionamento como correto ou anômalo. O sistema de detecção baseia-se na aquisição de dados através da corrente consumida pelo circuito a ser testado. Esses dados são processados e convertidos em imagens que são aplicadas no treinamento de um autocodificador. Este modelo é capaz de detectar, de forma satisfatória, quando existe alguma não conformidade para determinado sistema embarcado.

Palavras-chave: sistemas embarcados, detecção de anomalia, autocodificador.

Abstract

The constant need to optimize processes in industries has encouraged the use of embedded systems as a solution to problems in the most diverse segments. This trend has made the complexity of printed circuit boards (PCB) increasingly greater. On the other hand, this greater complexity generates a significant increase in the probability of failures for compact systems. The problem is that human skill and traditional testing methods have become insufficient to identify the presence of flaws in the system. Given this gap in the identification, this paper proposes the implementation on a compact platform of a non-invasive anomaly detection system, allowing the verification of embedded systems and the classification of their functioning as correct or anomalous. The detection system is based on the acquisition of data through the current consumed by the circuit to be tested. This data is processed and converted into images that are applied in the training of an autoencoder. This model is capable of satisfactorily detecting when there is any non-conformity for a given embedded system.

Keywords: embedded systems, anomaly detection, autoencoder.

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

PCI	*Placa de Circuito Impresso*
PCB	*Printed Circuit Board*
DUT	*Device Under Test*
ADC	*Analog-to-Digital Converter*
CPU	*Central Processing Unit*
GPU	*Graphics Processing Unit*
RNA	*Rede Neural Artificial*
PCA	*Principal Component Analysis*
OCC	*One Class Classification*
SIFT	*Scale-invariant Feature Transform*
LED	*Light-emitting Diode*
UART	*Universal Asynchronous Receiver/Transmitter*
ISR	*Interrupt Service Routine*
FW	*Firmware*
OK	*Okay*
NOK	*Not Okay*
GFLOPS	*Giga Floating-point Operations Per Second*
BSP	*Board Support Package*
RGB	*Red, Green, Blue*
MSE	*Mean Squared Error*

Sumário

1 Introdução

Os sistemas eletrônicos embarcados são sistemas computacionais compostos de parte lógica (*software* ou *firmware*) e parte física (*hardware)*, que estão cada vez mais presentes em nosso cotidiano.

A evolução tecnológica, o aumento da capacidade de *hardware* e o surgimento de novas aplicações impulsionadas pelo mercado e pelas necessidades de otimização de processos, incentivaram o uso desses sistemas para solucionar problemas nos mais diversos segmentos industriais.

No entanto, o aumento considerável na complexidade das placas de circuito impresso (PCI) desses sistemas embarcados exigiu a necessidade de mudanças no controle de qualidade dos circuitos eletrônicos desenvolvidos.

Os métodos tradicionais de testagem para encontrar defeitos de *hardware* e *software* em sistemas embarcados se tornaram ineficientes e a necessidade de atender o alto nível de personalização exigido pelos clientes se tornou cada vez mais significativa para os fabricantes desses dispositivos.

Como afirma Gilchrist (2016), um dos princípios básicos da Indústria 4.0 é conectar sistemas, máquinas e unidades de trabalho para criar redes inteligentes ao longo da cadeia de valor que podem funcionar separadamente e controlar umas às outras de forma autônoma, mas de maneira coesa.

Dessa forma, o desenvolvimento de metodologias que facilitem o processo de fabricação e validação da qualidade de funcionamento e segurança desses sistemas embarcados passou a ser uma premissa indispensável para o ramo da manufatura avançada.

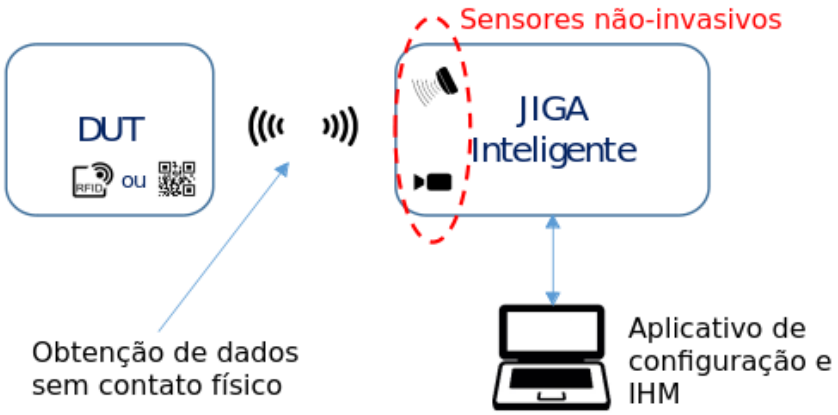
Facilitar esse processo significa reduzir o tempo de projeto e desenvolvimento, reduzir os custos e gastos recorrentes, garantir a capacidade de adaptação a cada variação do produto a ser manufaturado e, na medida do possível, utilizar métodos não invasivos para obter esses resultados.

Para Johnson (2018), o teste não invasivo pode neutralizar o custo cada vez maior de tecnologias de teste invasivas. Por ser orientado por software, esse tipo de teste é extremamente flexível e muito adaptável. Em particular, pode ser bastante eficiente quando os projetos de novos produtos estão passando do desenvolvimento para a fabricação de alto volume.

1.1 Contextualização

Este projeto consiste em analisar o funcionamento de um sistema inteligente de teste - que representa uma parte de um projeto de pesquisa maior e tem aplicação principal no âmbito da manufatura avançada (G. de OLIVEIRA, 2020) - com o objetivo de construir o sistema de testes em uma plataforma compacta. Na indústria, esse tipo de equipamento é denominado de Jiga de teste. Tal Jiga é capaz de realizar testes automatizados de circuitos eletrônicos, além de monitorar e coletar dados durante todo o processo.

Figura 1: Esquematização do processo de teste.



Fonte: Adaptado de G. de OLIVEIRA (2020).

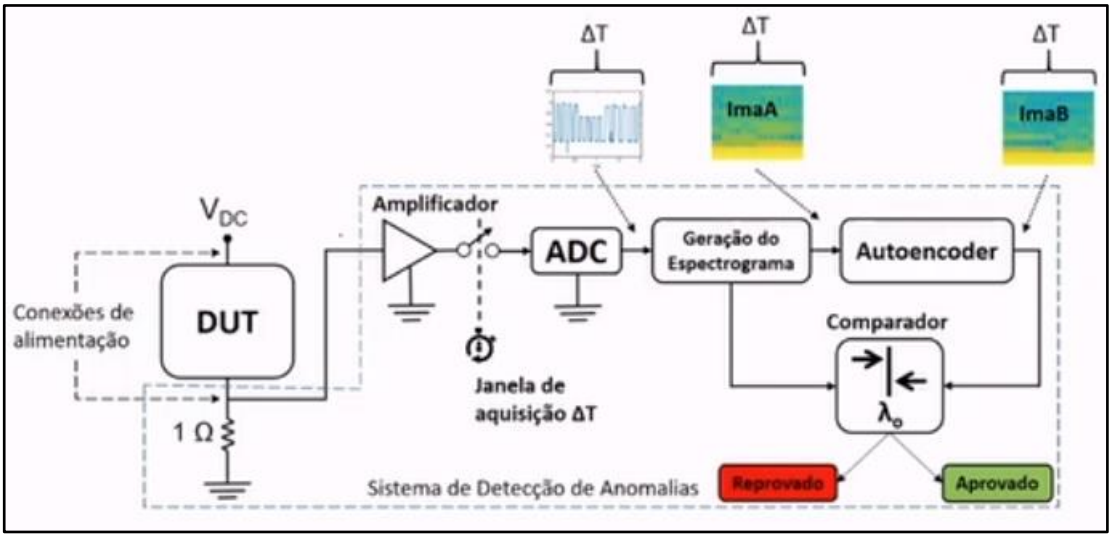
A solução proposta é focada numa solução não-invasiva, com o mínimo de contato lógico entre o dispositivo testado (DUT - *Device Under Test*) e o sistema de teste (Jiga de testes).

As principais características desse sistema são:

1. Alta confiabilidade;
2. Adaptabilidade a diferentes versões de placas;
3. Mínimo de invasibilidade possível na realização dos testes;

4. Capacidade de coletar dados para acompanhamento do produto ao longo de sua vida útil.

Figura 2: Detalhamento do sistema de teste.



Fonte: Adaptado de G. de OLIVEIRA (2020).

A Figura 2 resume a solução do sistema de detecção proposto. Utilizando a derivação de um resistor é possível realizar a medição, de maneira indireta, da corrente consumida em função do tempo. Depois de amplificar e coletar a corrente de acordo com a definição da janela de aquisição, que é um parâmetro ajustável, obtém-se um espectrograma e uma imagem referente à assinatura da corrente.

Para fazer a detecção de anomalias, é utilizado um autocodificador. Esse autocodificador recebe uma informação na entrada *X* (imagem referente à assinatura da corrente) e tem a função de reduzir a dimensionalidade dos dados e extrair características para gerar um código da informação utilizada na entrada em sua camada central. Neste contexto, Chollet (2016) afirma que os autocodificadores são uma técnica auto-supervisionada, uma instância específica de aprendizagem supervisionada onde os alvos são gerados a partir dos dados de entrada.

Segundo Alfeo et al. (2020), um autocodificador devidamente treinado é capaz de reconstruir corretamente entradas não anômalas. Como tal, a probabilidade de que uma determinada entrada seja uma anomalia pode ser avaliada pelo erro de reconstrução do autocodificador.

A vantagem de utilizar autocodificadores para este tipo de solução é a possibilidade de utilizar apenas uma classe para treino, no caso, a classe não anômala, já que seria muito difícil prever todos os cenários que poderiam ser classificados como anomalia no dispositivo. De acordo com Mujeeb et al. (2019), um autocodificador profundo treinado usando apenas imagens não anômalas pode aprender a extrair recursos que são diferentes dos recursos de uma imagem que é defeituosa.

Considerando que o autocodificador é treinado com a classe OK (não anômala), se a entrada for uma amostra *Xok* dessa mesma classe, na saída terá uma amostra reconstruída *X'ok* que será muito parecida com a imagem de entrada. Por outro lado, se a entrada for uma amostra *Ynok* que pertence à classe NÃO-OK (anômala), então a imagem reconstruída *Y'nok* vai ser muito diferente da imagem de entrada. Baseando-se no erro de reconstrução das imagens é possível classificar se o *firmware* é anômalo ou não, reprovando ou aprovando o DUT.

1.2 Objetivos

Este trabalho tem como objetivo estudar e aplicar técnicas de aprendizado de máquina e processamento de imagens no desenvolvimento do sistema, utilizando como principais ferramentas a linguagem de programação Python, a biblioteca de rede neural Keras rodando em conjunto com a biblioteca TensorFlow para desenvolvimento do autocodificador e sistema operacional Ubuntu 18.04.

Além disso, será avaliada a eficiência do treinamento do autocodificador ao utilizar um processador comum ou uma GPU dedicada com CUDA (COOK, 2012) para utilizar o alto potencial de processamento paralelo proporcionado por uma placa de vídeo, visto que o processo pode demandar muito tempo e recurso, dependendo da quantidade de amostras utilizadas e da quantidade de camadas utilizadas para codificar e decodificar as informações no autocodificador. A análise com GPU será realizada utilizando a plataforma da NVIDIA, Jetson Nano Developer Kit, que permite a execução paralela de redes neurais para aplicações com classificação de imagens.

Figura 3: NVIDIA Jetson Nano Developer Kit.

Fonte: NVIDIA (2019).

Por fim, é realizada uma análise da viabilidade de expandir o sistema de teste para classificação de diferentes plataformas compactas (portáteis), mantendo a alta confiabilidade do sistema.

1.3 Resultados e Impactos Esperados

O sistema de teste disponibiliza uma forma de testar sistemas eletrônicos embarcados para a Indústria 4.0 com alta adaptabilidade a diferentes plataformas e que não necessite ou minimize a interferência humana para classificação de anomalias.

Este trabalho visa à identificação da melhor metodologia para realizar o treinamento do autocodificador, assim como determinar os parâmetros que mais se adequam às diferentes plataformas, buscando a melhor forma de classificar se o sistema embarcado possui anomalia ou não.

Além disso, a coleta e o monitoramento dos dados gerados durante todo o processo poderão servir para destrinchar a classe anômala única em subclasses que possibilitem a identificação do tipo de anomalia encontrado no dispositivo que está sendo testado, e, dessa forma, o sistema ser capaz de apresentar possíveis soluções já conhecidas para o defeito classificado naquela plataforma.

O resultado esperado é a implementação da plataforma de testes e a geração de documentação técnica que viabilize sua replicação em um cenário de aplicação

real. O principal impacto diz respeito à eficiência no desenvolvimento da solução de testes de forma fisicamente compacta e financeiramente atrativa.

1.4 Estrutura da Monografia

Além deste capítulo, o trabalho está dividido em mais 4 capítulos:

- Capítulo 2 - Referencial Teórico: descreve conceitos usados como base para o desenvolvimento do trabalho.
- Capítulo 3 - Metodologia: expõe de que maneira foi realizado o estudo, a aquisição de dados e a extração das informações utilizadas para modelagem do sistema de detecção.
- Capítulo 4 - Desenvolvimento: apresenta a configuração de ambiente necessária para o desenvolvimento e a validação do trabalho, além de mostrar particularidades encontradas durante o processo de implementação.
- Capítulo 5 - Considerações Finais: analisa o impacto dos resultados obtidos e cita possíveis trabalhos futuros.

2 Referencial Teórico

Neste capítulo é apresentada a base teórica necessária ao entendimento de seções posteriores deste trabalho. Aqui são discutidos os fundamentos dos métodos de aprendizagem das redes neurais artificiais, as particularidades dos autocodificadores junto aos conceitos de codificação, decodificação e detecção de anomalias.

2.1 Conceitos chave

2.1.1 Rede Neural Artificial

O conceito por trás da metodologia de RNA (Rede Neural Artificial) está diretamente relacionado à modelagem do funcionamento da rede de neurônios do cérebro humano. Baseando-se nesse princípio, a ideia era tornar as máquinas capazes de realizar tarefas complexas como reconhecimento de padrões, previsão e generalização com elevada efetividade.

McCulloch & Pitts (1943) propuseram um modelo matemático de rede neural simples usando circuitos elétricos para descrever como os neurônios do cérebro poderiam funcionar. Esse modelo contribuiu fortemente com as discussões acerca das áreas da computação que buscavam levar comportamentos inspirados na natureza para dispositivos artificiais.

De acordo com Haykin (2001), para obter um bom desempenho, as redes neurais utilizam um grande número de interconexões de unidades de computação simples denominadas "neurônios" ou unidades de processamento que podem não apenas processar informações de forma distribuída, mas também integrar conhecimento por meio de informações extraídas do ambiente através de um processo de *aprendizagem* ou *treinamento*. Os pesos de conexão entre os neurônios, chamados de pesos sinápticos, são usados para armazenar conhecimento e disponibilizar as informações adquiridas em um momento futuro. Além disso, cada

unidade de processamento possui uma função de ativação que tem como argumento a soma ponderada dos sinais de entrada, para determinar sua saída. A função de ativação representa o efeito que a entrada e o estado atual de ativação exercem na definição do próximo estado da unidade.

Uma das principais vantagens em utilizar RNAs é a capacidade que esses dispositivos possuem de se adequar à situação em que estão inseridos. Essa característica deve-se à sua propriedade de inferir relações não lineares complexas.

2.1.1.1 Redes *Feedforward* de Múltiplas Camadas

Uma das possíveis arquiteturas utilizadas para estruturar os neurônios de uma RNA. É caracterizada pela presença de uma ou mais camadas intermediárias ou escondidas (camadas em que os neurônios são efetivamente unidades processadoras, mas não correspondem à camada de saída), aumentando assim, o poder de processamento não-linear e de armazenamento da rede. As saídas dos neurônios de cada camada são utilizada como entradas para a camada seguinte.

Essa classe *feedforward* (propagação direta) de múltiplas camadas, são geralmente treinadas utilizando o algoritmo de *error backpropagation* (retropropagação do erro).

2.1.1.2 Métodos de Aprendizagem

Uma rede neural aprende, basicamente, através de um processo iterativo de ajuste de pesos e limiares. De acordo com Haykin (1998), o processo de aprendizagem (treinamento) de uma rede neural é determinado pela forma com que seus parâmetros se adaptam aos estímulos do ambiente em que a rede está inserida. Os tipos de aprendizado são divididos em supervisionado e não supervisionado.

No aprendizado supervisionado, o modelo é treinado utilizando dados bem rotulados, ou seja, as variáveis de entrada e saída são previamente conhecidas e fornecidas ao modelo. Basicamente, esse modelo usa os dados de treinamento para aprender a ligar os dados de entrada e saída. Para criar, dimensionar e implementar esse tipo de aprendizado numa rede neural é necessário tempo e conhecimento técnico de uma equipe de cientistas de dados para garantir que os modelos sejam capazes de fornecer as informações corretamente. Esse método geralmente é

altamente preciso e confiável, mas pode ter seu desempenho comprometido se a quantidade de dados for muito grande.

O aprendizado não supervisionado é uma técnica que permite que o modelo trabalhe por conta própria para descobrir informações, lidando principalmente com dados não rotulados ou desconhecidos. Esse tipo de aprendizado permite que tarefas ainda mais complexas sejam realizadas. Contudo, o aprendizado não supervisionado é computacionalmente complexo e pode ser mais imprevisível se comparado a outros métodos de aprendizagem profunda.

2.1.2 Análise de Componente Principal (PCA)

Análise de componente principal, introduzida por Pearson (1901), ou PCA, é uma técnica de análise estatística usada para compressão, visualização e classificação de dados. A ideia central é reduzir a dimensionalidade de grandes conjuntos de dados, transformando um grande conjunto de variáveis em um menor, preservando o máximo possível das informações.

A PCA faz uma transformação linear das variáveis originais em um novo conjunto de variáveis que de modo que as variáveis se tornem independentes e sem redundância e isso, no geral, traz consigo uma perda de precisão, mas torna o problema mais simples de ser explorado e analisado.

A análise é feita a partir da matriz de covariância dos dados, que mede o grau de interdependência entre duas variáveis aleatórias. Em seguida, os autovetores e autovalores são calculados e a matriz de projeção com os autovetores correspondentes aos maiores autovalores são utilizados para projetar o novo conjunto.

A técnica é uma forma de aprendizagem não supervisionada, pois depende inteiramente dos próprios dados de entrada, sem referência aos dados de destino correspondentes, mas nem sempre é apropriada, pois recursos com baixa variação podem ter alta relevância preditiva para uma determinada aplicação, além disso, a técnica é limitada a componentes lineares.

2.1.3 Autocodificadores

Um autocodificador é um algoritmo de aprendizado não supervisionado que tem o objetivo de reproduzir na sua saída uma reconstrução da própria entrada, preservando os aspectos mais relevantes dos dados. Apesar de ser considerado um algoritmo de aprendizado não supervisionado, os autocodificadores são guiados por uma instância específica de treinamento supervisionado, denominada de auto-supervisionada, como mencionado na seção de contextualização do Capítulo 1 deste trabalho.

O algoritmo é constituído de duas partes principais que são o codificador e o decodificador. O codificador é uma função de extração de características, que mapeia o conjunto de treinamento para uma representação latente, geralmente com redução de dimensão, e o decodificador é uma função de reconstrução, que mapeia a representação produzida pelo codificador de volta para o espaço original. A finalidade do treinamento é definir parâmetros no codificador e no decodificador tal que o erro de reconstrução seja minimizado.

Os autocodificadores são treinados para dados específicos, ou seja, eles só funcionarão corretamente com dados semelhantes aos dados com os quais foram treinados. Por conta da redução de dimensão, também existirão perdas na precisão dos dados em comparação com as entradas originais. Além disso, um autocodificador sem funções de ativação tem capacidade semelhante a PCA, mas, em contrapartida, o algoritmo não se limita a componentes lineares.

2.1.4 Detecção de anomalias

As anomalias são definidas como eventos que se desviam do padrão, raramente acontecem e não seguem o resto do "padrão". Por exemplo, um dispositivo defeituoso em uma fábrica de sistemas embarcados.

Os algoritmos de detecção de anomalias podem ser divididos em duas subclasses: detecção de *outlier* (ponto fora da curva) e detecção de novidade.

Para a detecção de *outlier*, considerando que o conjunto de dados de entrada contém exemplos de eventos normais e eventos anômalos, esses algoritmos buscam ajustar as regiões dos dados de treinamento onde os eventos são mais concentrados,

desconsiderando e isolando os eventos anômalos. Dessa forma, o algoritmo agrupa os dados em classes anômalas ou não. Essa subclasse é frequentemente treinada de forma não supervisionada, ou seja, sem rótulos.

Já para a detecção de novidade, os algoritmos têm apenas os pontos relacionados a um evento normal, ou seja, nenhum evento de anomalia, durante o seu treinamento. Dessa forma, cabe ao algoritmo prever se um determinado conjunto de dados é uma anomalia ou não no momento do teste. Essa subclasse é treinada de forma supervisionada, usando rótulos para os conjuntos de dados padronizados.

2.2 Trabalhos Relacionados

A ideia de utilizar métodos de detecção de anomalias utilizando autocodificadores já vem sendo explorada por outros autores. Esta seção mostra soluções que, de certa forma, se assemelham com a ideia deste trabalho.

2.2.1 Uso de um autocodificador no projeto de um detector de anomalias para manufatura avançada

Segundo o paradigma da manufatura avançada, a análise da série temporal de ativos com uma abordagem de aprendizado de máquina pode prevenir efetivamente contratempos não planejados na linha de produção, através da detecção de condições operacionais anômalas (ALFEO et al, 2020).

Utilizando sinais de série temporal coletadas de máquinas de lavar, o sistema extrai características das amostras onde o sinal fica acima de um limiar e utiliza essas informações para treinar um autocodificador.

A reconstrução do dado na saída é comparada com o dado na entrada, e o erro é enviado para um discriminador baseado em uma heurística geral, que transforma esse valor em uma probabilidade e indica automaticamente se a máquina está operando normalmente ou com alguma anomalia.

Os resultados foram apresentados em termos de erro de treinamento, mostrando que o conceito da pesquisa tinha uma relevância grande para a área de estudo, mas não foram realizados testes de detecção de anomalia com o modelo obtido através do treinamento do sistema.

2.2.2 Abordagem de aprendizagem de recursos baseada em classe única para detecção de defeitos usando autocodificadores profundos

Neste artigo, é proposta uma abordagem baseada em aprendizado profundo utilizando autocodificadores para extração das características, que é capaz de detectar diferentes defeitos em placas de circuito impresso (PCI) sem usar nenhuma amostra defeituosa durante o treinamento.

O método usado, onde apenas amostras de uma classe (amostras sem defeito) são disponíveis para treinamento é chamado de *One Class Classification* (OCC) e também serviu de inspiração para o desenvolvimento deste trabalho.

Os resultados foram obtidos através de uma comparação algorítmica de imagens chamada SIFT (*Scale-invariant Feature Transform*), (LOWE et al, 2004), que define o nível de similaridade de duas imagens. Em comparação com outros métodos de processamento de imagem, a utilização do SIFT se mostrou mais flexível e eficaz, sendo capaz de detectar diferentes tipos de defeitos com personalização mínima.

3 Metodologia

Este capítulo descreve o método de pesquisa utilizado e como foram elaboradas e executadas as diferentes etapas da confecção deste trabalho a fim de alcançar os objetivos descritos no Capítulo 1.

3.1 Qualificação da Metodologia

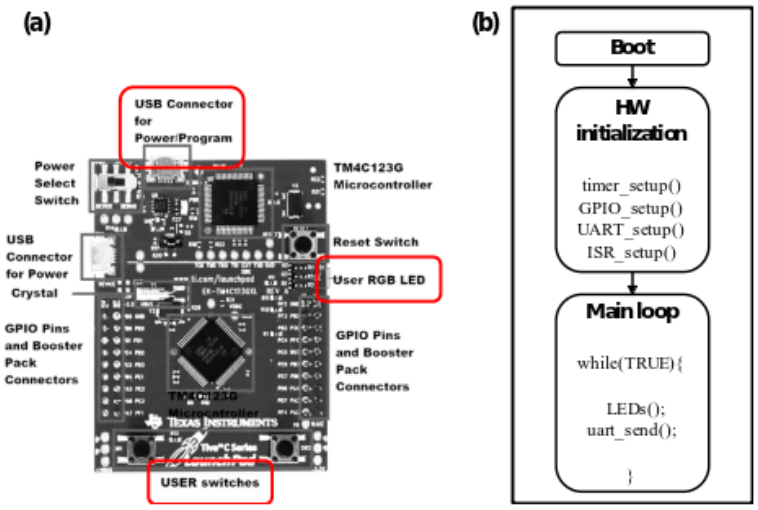
Como pode ser observado no Capítulo 2, a fundamentação teórica necessária para o entendimento deste trabalho é voltada para o estudo de técnicas de aprendizado de máquina. Trata-se de uma pesquisa bibliográfica, elaborada a partir de conceitos já publicados em livros, artigos, manuais e outros.

3.2 Projeto e construção do *Device Under Test* (DUT)

O sistema de teste experimental foi construído utilizando-se a placa Tiva C da Texas Instruments (T.I., 2013). O desenvolvimento do conjunto de *firmware* da placa foi elaborado com base em soluções de *firmware* de sistemas embarcados clássicos.

Figura 4: Placa Tiva C - Texas Instruments.

(a) *hardware* da placa Tiva C; (b) esquematização genérica de sistema embarcado.



Fonte: Adaptado de G. de OLIVEIRA (2020).

Basicamente, o *firmware* funciona da seguinte maneira: após o ligamento da placa, é realizada a inicialização do *hardware* e logo em seguida o dispositivo entra num *loop* infinito, executando duas funcionalidades: uma relacionada à interface de LED (*Light-emitting Diode)* e outra à interface serial UART (*Universal Asynchronous Receiver/Transmitter)*, cujo envio é feito por uma rotina de interrupção *ISR_setup* (*Interrupt Service Routine)*.

Código 3.1: Funcionamento distinto entre múltiplas versões de *firmware.*

```
while (1) {
    switch (ui8FWVersion) {
        case 0:
            // Piscando LED alternadamente em uma cadência: G -> G+B -> R+G+B
            ui8LEDCounter++;
            SysCtlDelay(100);
            if (ui8LEDCounter < 100){
                // ...
            } else if (ui8LEDCounter < 200) {
                // ...
            } else if (ui8LEDCounter < 300) {
                // ...
            } else {
                ui8LEDCounter = 0;
            }
            break;

        case 1:
            // Piscando LED alternadamente em uma cadência: G -> G+B -> R+G+B
            ui8LEDCounter++;
            SysCtlDelay(100);
            if (ui8LEDCounter < 13){
                // ...
            } else if (ui8LEDCounter < 18) {
                // ...
            } else if (ui8LEDCounter < 23) {
                // ...
            } else {
                ui8LEDCounter = 0;
            }
            break;
    }
}
```

FW_OK

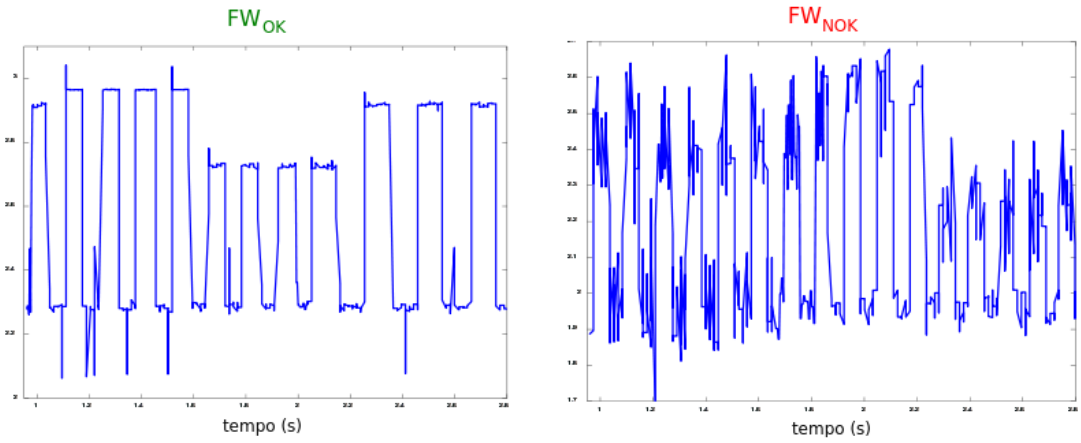
FW_NOK

Fonte: Adaptado de G. de OLIVEIRA (2020).

O Código 3.1 representa um exemplo de *firmware* com duas possíveis versões. Para os dois casos possíveis, o código obedece a uma sequência de acendimento dos LEDs que segue o mesmo padrão: acende o LED verde, em seguida o LED verde e o LED azul e, por fim, acende os três LEDs juntos. Os valores representados nas estruturas condicionais definem a temporização do acendimento e apagamento dos LEDs.

Para esse exemplo, a versão do *case 0* é considerada como um *firmware* (FW) com funcionamento correto e dentro do esperado (FW OK). Já a versão do *case 1* apresenta modificações nos valores das estruturas condicionais em comparação a versão do *case 0*. Dessa forma, como o comportamento dos LEDs do *firmware* do *case 1* vai ter uma cadência diferente em seu acendimento e apagamento, ele pode ser considerado como inadequado ou anômalo (FW NOK) pelo sistema.

Figura 5: Corrente adquirida para duas versões distintas de *firmware.*



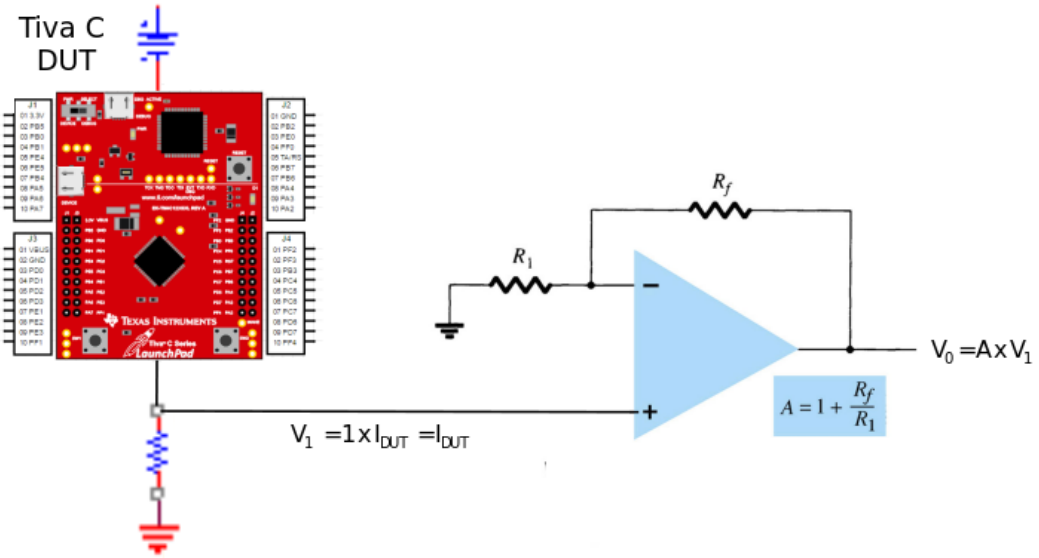
Fonte: Adaptado de G. de OLIVEIRA (2020).

É possível verificar essa diferença nos comportamentos entre as duas versões ao adquirir a corrente consumida pelo sistema de acordo com o *firmware* que está sendo executado, e dessa forma, determinar se existe anomalia ou não para aquela versão.

3.3 Aquisição dos Dados

Os dados utilizados para realização desta pesquisa foram coletados a partir da JIGA de testes mencionada no Capítulo 1. Foram utilizadas duas placas Tiva C, uma para ser o DUT e outra para fazer a aquisição dos dados.

Figura 6: Circuito para aquisição de corrente consumida pela Tiva C (DUT).

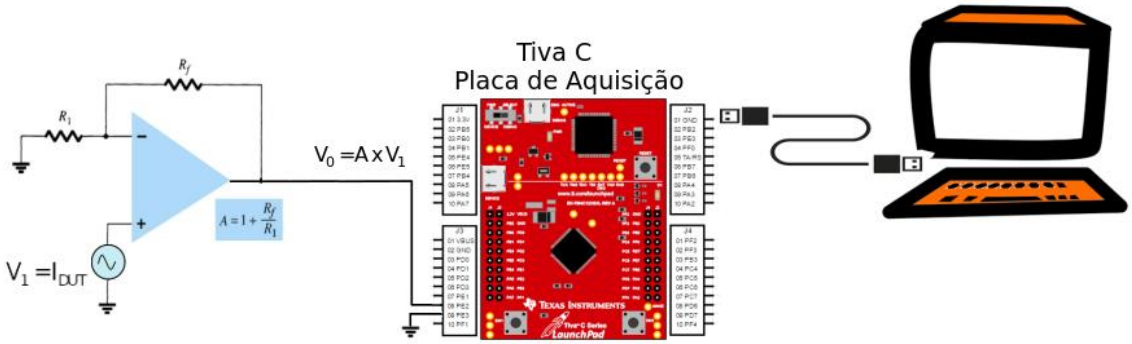


Fonte: Adaptado de G. de OLIVEIRA (2020).

Para a placa Tiva C utilizada como DUT, um resistor de 1 Ω foi acoplado em série. Pela Lei de Ohm, medindo a tensão é possível encontrar a corrente consumida. Pelo fato da tensão medida ser muito baixa, foi necessário amplificá-la, utilizando um amplificador operacional de baixo ruído e de uso geral com um ganho de tensão de aproximadamente 50.

Em seguida, essa tensão amplificada foi adquirida por outra placa Tiva C (placa de aquisição) por meio da sua entrada de conversão analógico-digital (ADC - *Analog-to-Digital Converter*).

Figura 7: Conversão da informação adquirida para posterior análise.



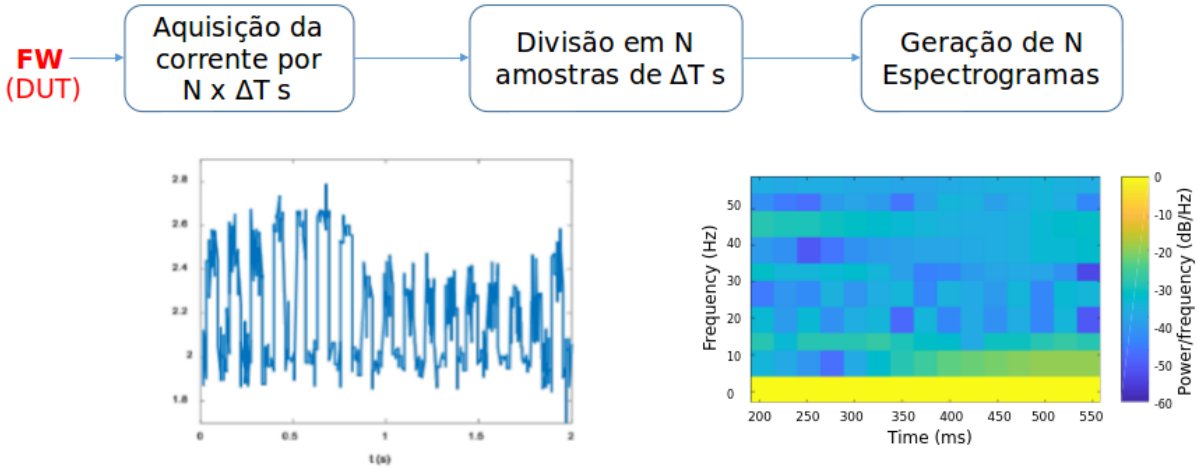
Fonte: Adaptado de G. de OLIVEIRA (2020).

A informação adquirida é, então, enviada para um computador para ser analisada utilizando MATLAB e Python.

3.3.1 Geração de Amostras

Primeiramente é feita a aquisição da corrente do *firmware* executado no dispositivo que está sendo testado durante um intervalo de tempo especificado por N x ΔT. Em seguida, toda a amostra coletada é dividida em N amostras de ΔT segundos, onde ΔT é a janela de aquisição de dados. O próximo passo é gerar um espectrograma com a mesma largura da corrente adquirida em função do tempo.

Figura 8: Transformação de corrente consumida em espectrograma.



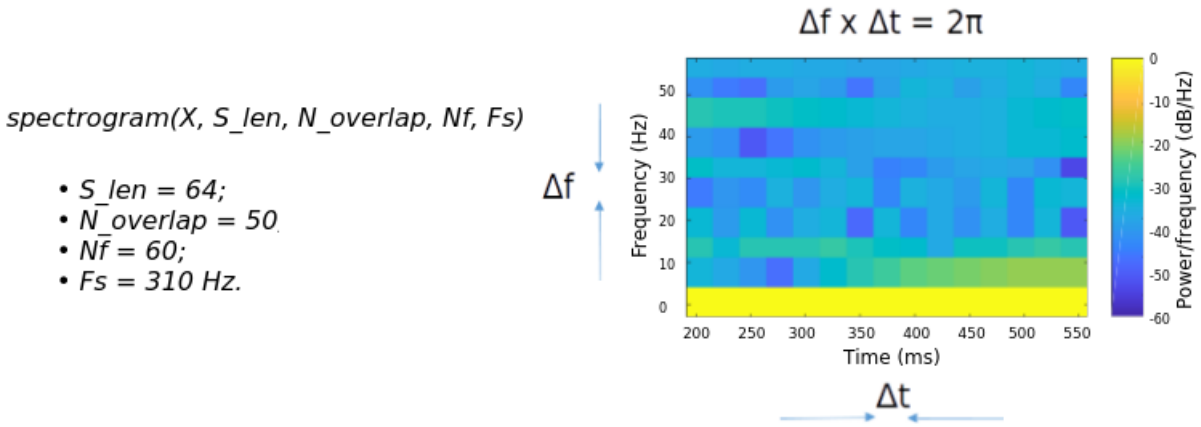
Fonte: Adaptado de G. de OLIVEIRA (2020).

3.3.2 Geração de Espectrogramas

Para gerar os espectrogramas, foi utilizada a função *spectrogram* do MATLAB por conta da simplicidade de visualização que a interface da ferramenta oferece, facilitando o processo de ajuste de parâmetros, como a variação da frequência e a variação do tempo dos experimentos.

Para a construção do espectrograma é necessário assumir um princípio da incerteza onde Δf x Δt, além dos valores atribuídos aos parâmetros da função geradora.

Figura 9: Parâmetros para geração de espectrograma.

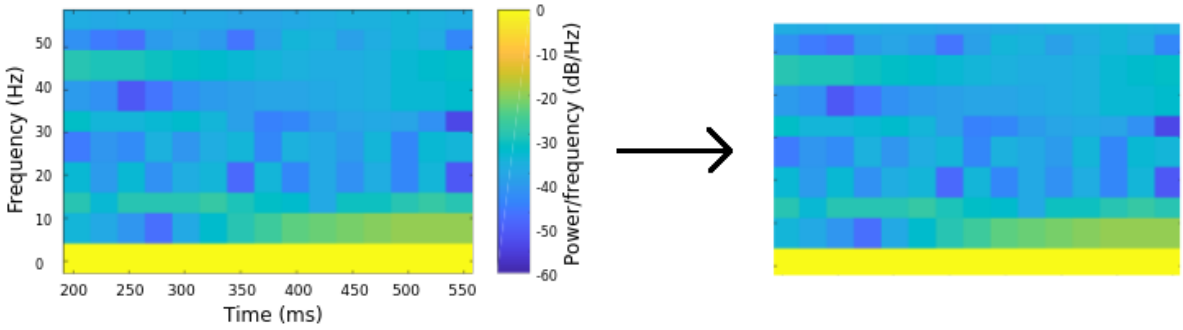


Fonte: Adaptado de G. de OLIVEIRA (2020).

Os valores utilizados como parâmetros da função *spectrogram* para esse experimento estão evidenciados na Figura 9. Vale ressaltar a importância da escolha do número de frequências (Nf), que determina o número de *bins* utilizados para definir o espectrograma, pois quanto maior a quantidade de frequência, mais largo vai ser o Δf e menor o Δt e vice-versa. Para esse experimento foram utilizados 60 *bins*. Além disso, a taxa de aquisição (Fs) é limitada pelo ADC da placa Tiva C em 310 Hz.

Pelo fato de o sinal de interesse do experimento ter frequências que chegavam até 60 Hz, é possível, durante o processo de geração, utilizar apenas o corte da imagem do espectrograma limitada à frequência de 60 Hz, como uma forma de filtrar ruídos do sinal acima desse limite.

Figura 10: Transformação de espectrograma em imagem.



Fonte: Adaptado de G. de OLIVEIRA (2020).

As imagens geradas durante essa etapa da pesquisa foram separadas em diferentes classes, onde apenas uma delas era considerada com funcionamento correto e as outras apresentavam algum tipo de anomalia em seu firmware.

Para identificação do melhor método a ser utilizado para o sistema de detecção de anomalias foi realizada uma análise exploratória entre alguns algoritmos.

Tabela 1: Versões de *firmware* e características para treinamento.

Designation	FW_{OK}	FW_{NOK1}	FW_{NOK2}	FW_{NOK3}	FW_{NOK4}	FW_{NOK5}
Operation	Normal (anomaly free)	LED activation period change	UART multiple character transmission	Inverted LED activation order	Interrupt service routine delay	Red LED defect simulation
Total samples	14599	13909	2000	2000	2000	2000
Autoencoder training	10079	NA	NA	NA	NA	NA
Autoencoder validation	2520	NA	NA	NA	NA	NA
Autoencoder testing	2000	2000	2000	2000	2000	2000
Random forest training	12599	11909	none	none	none	none
Random forest testing	2000	2000	none	none	none	none
SIFT testing	2000	2000	none	none	none	none

Fonte: Adaptado de G. de OLIVEIRA (2020).

Na tabela 1, pode-se observar as diferentes versões de *firmware* utilizadas na análise exploratória. Vale ressaltar que para todos os testes foi utilizada a mesma janela de aquisição $\Delta T = 2s$.

Considerando a falta de disponibilidade e a dificuldade de criar modelos que sejam capazes de classificar a corretude do funcionamento dos sistemas embarcados, visto que comportamentos não mapeados poderiam não ser interpretados como uma anomalia para o sistema, foi feita uma comparação entre as técnicas listadas.

Ao comparar o treinamento entre as diferentes técnicas, é possível identificar que para o autocodificador só é necessário utilizar a classe sem anomalias, diferentemente dos outros métodos, que precisam de pelo menos uma classe anômala para seu treinamento.

Dessa forma, como a ideia geral é provar o conceito de que o sistema de detecção de anomalias nesses sistemas compactos é capaz de identificar se existe anomalia, ou não, no funcionamento do dispositivo, essa característica dos

autocodificadores foi imprescindível para a escolha desta técnica como a mais adequada para o desenvolvimento da solução desta pesquisa.

4 Desenvolvimento

Este capítulo descreve como foi realizada a configuração do ambiente para o desenvolvimento, treinamento, testagem e validação do sistema de detecção de anomalias utilizando autocodificadores e como as dificuldades encontradas foram contornadas para possibilitar a prova de conceito desta pesquisa.

4.1 Configuração do Ambiente

Para tornar o computador capaz de implementar um autocodificador é necessário realizar uma série de configurações utilizando bibliotecas projetadas especificamente para criação e treinamento de redes neurais, que tem como objetivo detectar e decifrar padrões e correlações de forma análoga à forma como os humanos aprendem e raciocinam.

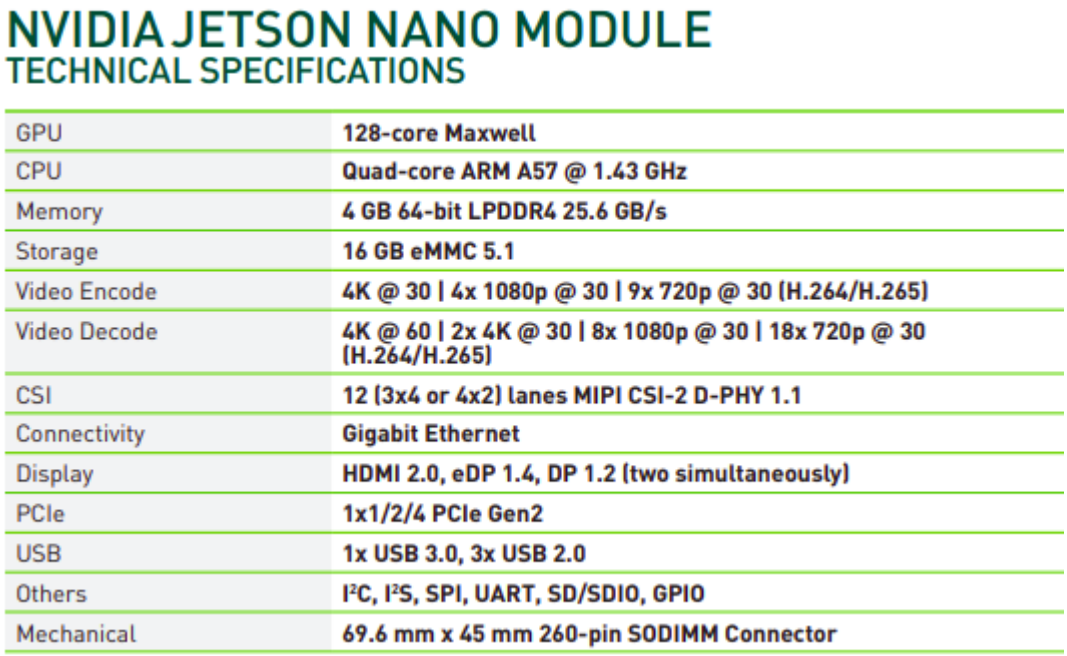
As principais plataformas utilizadas neste trabalho foram Keras (CHOLLET, 2015) e TensorFlow (ABADI et al, 2015), que possibilitam a criação de modelos de aprendizado de máquina, em conjunto com a linguagem de programação Python 3 e o poder da visão computacional oferecido pela ferramenta OpenCV (BRADSKI, 2000). Além desses principais componentes, a instalação das dependências para essas bibliotecas também é de extrema importância para garantir o funcionamento correto e esperado que essas bibliotecas oferecem.

4.1.1 NVIDIA Jetson Nano Developer Kit

A escolha do computador utilizado para realizar a implementação do auto codificador foi baseada na documentação disponível sobre o módulo NVIDIA Jetson Nano (Jetson Nano, 2019), que oferece até 472 GFLOPS de computação acelerada, permitindo a execução de muitas redes neurais em paralelo, com suporte a diversos *frameworks* de inteligência artificial.

O *system-on-module* é alimentado por NVIDIA Maxwell GPU com 4 GB de memória, o que viabiliza o processamento em tempo real de entradas de alta resolução.

Figura 11: Especificações técnicas do módulo NVIDIA Jetson Nano.



NVIDIA JETSON NANO MODULE
TECHNICAL SPECIFICATIONS

GPU	128-core Maxwell
CPU	Quad-core ARM A57 @ 1.43 GHz
Memory	4 GB 64-bit LPDDR4 25.6 GB/s
Storage	16 GB eMMC 5.1
Video Encode	4K @ 30 | 4x 1080p @ 30 | 9x 720p @ 30 (H.264/H.265)
Video Decode	4K @ 60 | 2x 4K @ 30 | 8x 1080p @ 30 | 18x 720p @ 30 (H.264/H.265)
CSI	12 (3x4 or 4x2) lanes MIPI CSI-2 D-PHY 1.1
Connectivity	Gigabit Ethernet
Display	HDMI 2.0, eDP 1.4, DP 1.2 (two simultaneously)
PCIe	1x1/2/4 PCIe Gen2
USB	1x USB 3.0, 3x USB 2.0
Others	I²C, I²S, SPI, UART, SD/SDIO, GPIO
Mechanical	69.6 mm x 45 mm 260-pin SODIMM Connector

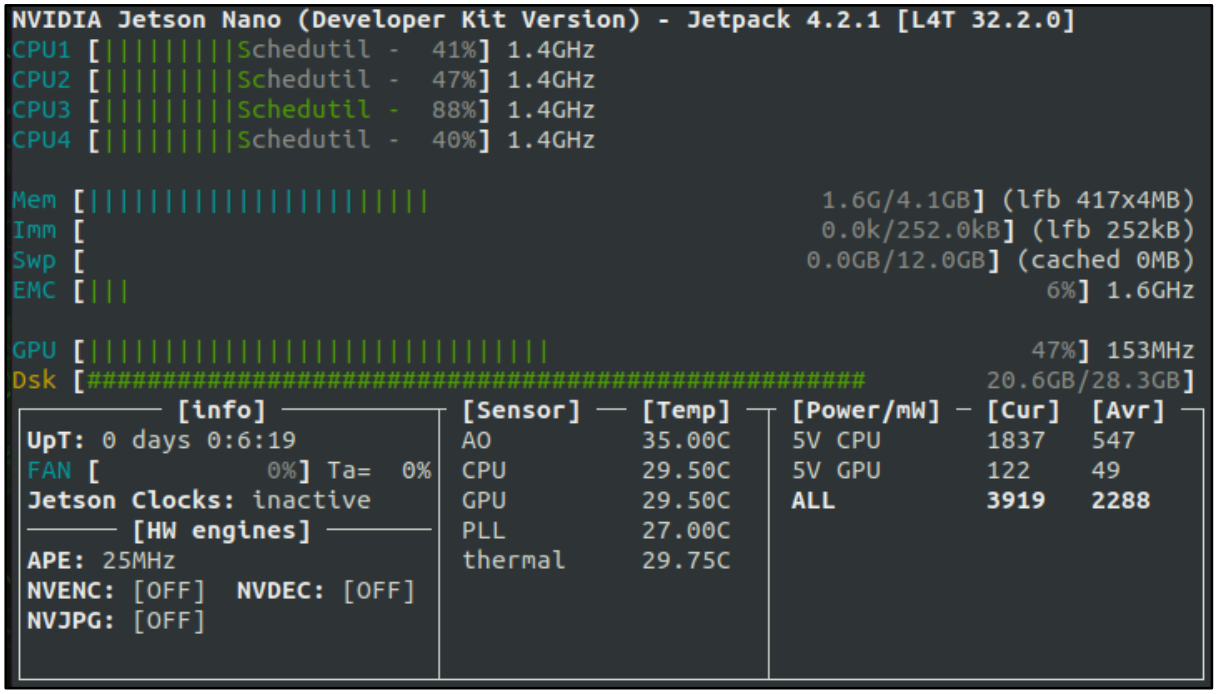
Fonte: Jetson Nano (2019).

4.1.1.1 JetPack & CUDA

A placa Jetson Nano é compatível com NVIDIA JetPack, que inclui um *board support package* (BSP), sistema operacional Linux, NVIDIA CUDA, cuDNN e bibliotecas TensorRT utilizadas para aprendizagem profunda, visão computacional e computação GPU.

Para realizar a configuração de *setup* do módulo é necessário gravar a imagem do JetPack através de um cartão microSD, e dessa forma ser capaz de inicializar o sistema operacional.

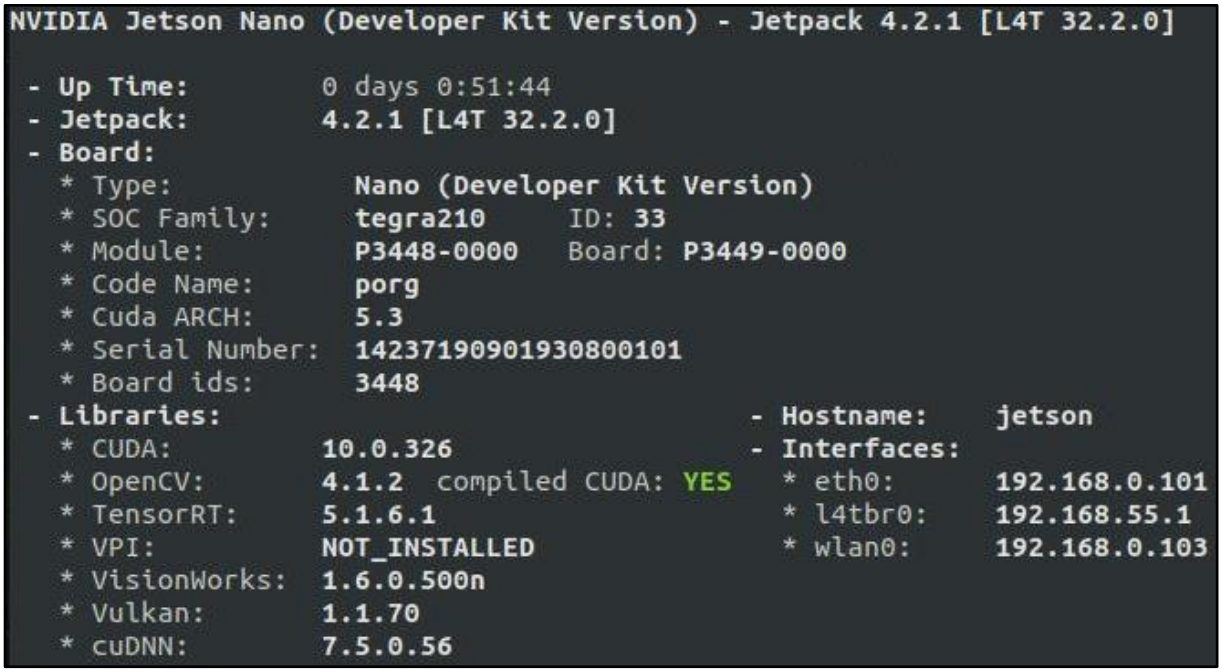
Figura 12: Status da placa Jetson Nano.



Fonte: Elaborada pelo autor.

Após a conclusão das etapas de *setup*, foram feitas todas as instalações necessárias para a infraestrutura dos algoritmos, como a integração do CUDA com OpenCV e a atualização do TensorFlow para uma versão específica que garante maior compatibilidade com o módulo.

Figura 13: Resumo das versões das ferramentas instaladas no módulo.



Fonte: Elaborada pelo autor.

4.2 Implementação do Autocodificador

O objetivo é treinar uma rede que possa aprender a reconstruir os dados de entrada, como visto na seção 2.1.4. Para isso, utilizaremos as imagens conseguidas após a etapa de geração de espectrograma da seção 3.3.2.

O código 4.1 mostra como é inicializada a sessão do *framework* TensorFlow, utilizando a opção *allow_growth=True* para especificar que a GPU só tente alocar apenas a quantidade de memória com base nas alocações de tempo de execução, começando com pouca memória e, conforme as sessões são executadas e mais memória da GPU é necessária, a região de memória necessitada para o processamento é estendida.

Código 4.1: Inicialização da sessão no TensorFlow.

```
import keras
import tensorflow as tf

config = tf.ConfigProto()
config.gpu_options.allow_growth=True
sess = tf.Session(config=config)
keras.backend.set_session(sess)
```

Fonte: Elaborada pelo autor.

Como as entradas do autocodificador são as imagens, é adequado utilizar redes neurais convolucionais como codificadores e decodificadores, visto que elas apresentam um desempenho muito melhor para este caso. As imagens filtradas do espectrograma têm 128 x 96 pixels (altura e largura) e 3 canais (RGB) para as cores.

Para atender às limitações do ambiente compacto (placa Jetson Nano) e suas restrições de recurso, sobretudo a memória RAM, o autocodificador originalmente utilizado em G. de OLIVEIRA (2020) precisou ser aperfeiçoado. A atividade de aperfeiçoamento do autocodificador representa uma das principais contribuições deste trabalho. O resultado final está ilustrado no Código 4.2, onde *input_img* é a imagem de entrada, *encoded* é a representação codificada da entrada e *decoded* é a reconstrução com perdas da entrada.

Código 4.2: Implementação do codificador e decodificador do modelo.

```
from keras import layers

input_img = keras.Input(shape=(96, 128, 3))

e = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(input_img)
e = layers.MaxPooling2D((2, 2), padding='same')(e)
e = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(e)
encoded = layers.MaxPooling2D((2, 2), padding='same')(e)

d = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(encoded)
d = layers.UpSampling2D((2,2))(d)
d = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(d)
d = layers.UpSampling2D((2,2))(d)
decoded = layers.Conv2D(3, (3, 3), activation='sigmoid', padding='same')(d)

autoencoder = keras.Model(input_img, decoded)
```

Fonte: Elaborada pelo autor.

O codificador consiste em uma pilha de camadas Conv2D e MaxPooling2D. As camadas Conv2D criam um *kernel* de convolução para produzir um tensor de saídas a partir da entrada. Já as camadas MaxPooling2D reduzem a dimensão da representação do espaço latente.

Para o decodificador, a pilha consiste em camadas de Conv2D e UpSampling2D, onde essas últimas fazem a operação inversa do MaxPooling2D e aumentam a dimensão da representação do espaço latente.

Na última linha do Código 4.2, é feita a atribuição do modelo que mapeia uma imagem de entrada para sua reconstrução, com as características contidas no codificador e no decodificador, na variável *autoencoder*.

4.2.1 Pré-processamento das imagens de entrada

Para gerar lotes de dados de imagem de tensor com aumento de dados em tempo real, é utilizada a função da biblioteca Keras para pré-processamento das imagens de entrada. Cada lote gerado, sendo ele para treino, teste ou validação tem seu tamanho especificado no código, sendo um parâmetro customizável para a quantidade de imagens desejada, levando em consideração a limitação de processamento do computador.

Código 4.3: Atribuição do tamanho do lote para cada classe de imagens.

```
from keras.preprocessing.image import ImageDataGenerator

Train_Generator_Batch_Size = 7000 # 7000 imagens (Treino)
Test_Generator_Batch_Size = 2000 # 2000 imagens (Teste)
Validation_Generator_Batch_Size = 2000 # 2000 imagens (Validação)
```

Fonte: Elaborada pelo autor.

No código 4.4, é mostrada a função que faz a geração do lote de dados referentes às imagens utilizadas para treinamento. Os parâmetros exibidos nas três primeiras linhas da função são referentes às dimensões da imagem 128 x 96 pixels (altura e largura), ao diretório onde as imagens estão salvas e ao tamanho do lote, que para esse caso seria de 7000 imagens como mostrado no Código 4.3.

Código 4.4: Função para pré-processamento das imagens de treino.

```
def train_images():
    img_width, img_height = 96, 128
    train_data_dir = './Train'
    batch_size = Train_Generator_Batch_Size

    train_datagen = ImageDataGenerator(
        rescale=1. / 255)

    train_generator = train_datagen.flow_from_directory(
        train_data_dir,
        target_size=(img_width, img_height),
        batch_size=batch_size,
        color_mode='rgb',
        class_mode='input',
        shuffle=True,
        seed=42)

    x =  train_generator
    return x[0][0], x[0][1]
```

Fonte: Elaborada pelo autor.

Em seguida, a função ImageDataGenerator é utilizada para re-escalar os pixels da imagem. *Rescale* 1./255 transforma o valor de cada pixel do intervalo [0, 255] para [0, 1]. Os benefícios provenientes dessa transformação servem para tratar as imagens da mesma maneira: o dimensionamento de todas as imagens para o mesmo intervalo [0, 1] fará com que as imagens contribuam de forma mais uniforme para a perda total.

Por fim, a função *flow_from_directory* do Keras utiliza esses parâmetros como argumentos e retorna uma tupla referente ao lote das imagens, junto com seus respectivos rótulos.

Ao finalizar o pré-processamento das imagens de treino, também é realizada a geração dos dados para as imagens de validação e teste. Como exibido no Código 4.5, as funções são muito parecidas com a utilizada para treino, mas diferem no diretório utilizado, que deve corresponder a pasta onde as imagens devem ser encontradas, e no argumento *shuffle* da função *flow_from_directory* que serve para randomizar a aquisição dos dados quando verdadeiro e para seguir uma ordem alfanumérica quando falso.

Código 4.5: Funções para pré-processamento das imagens de validação e teste.

```
def valid_images():
    img_width, img_height = 96, 128
    valid_data_dir = './Valid'
    batch_size = Validation_Generator_Batch_Size

    valid_datagen = ImageDataGenerator(
        rescale=1. / 255)

    valid_generator = valid_datagen.flow_from_directory(
        valid_data_dir,
        target_size=(img_width, img_height),
        batch_size=batch_size,
        color_mode='rgb',
        class_mode='input',
        shuffle=False,
        seed=42)

    x =  valid_generator
    return x[0][0], x[0][1]
```

```
def test_images():
    img_width, img_height = 96, 128
    test_data_dir = './Test'
    batch_size = Test_Generator_Batch_Size

    test_datagen = ImageDataGenerator(rescale=1. / 255)

    test_generator = test_datagen.flow_from_directory(
        test_data_dir,
        target_size=(img_width, img_height),
        batch_size=batch_size,
        color_mode='rgb',
        class_mode='input',
        shuffle=False,
        seed=42)

    x =  test_generator
    return x[0][0], x[0][1]
```

Fonte: Elaborada pelo autor.

4.2.2 Memória Virtual

O Jetson Nano por padrão tem 2 GB de memória *swap*. A memória virtual permite “memória extra” quando há alta demanda pela memória principal (física), trocando partes da memória principal para a secundária. Como o Jetson Nano tem uma quantidade relativamente pequena de memória (4 GB), isso pode ser muito útil.

Dessa forma, para realizar o pré-processamento foi verificada a necessidade de aumentar a memória virtual para 4 GB, para garantir que o pré-processamento do lote das imagens não sofra nenhum travamento durante sua execução, visto que um lote com muitas imagens pode comprometer a memória principal com exceções de falta de memória.

4.2.3 Compilando e treinando o autocodificador

Depois de pré-processados, os dados das imagens são utilizados para realizar a compilação e o treinamento do autocodificador. Para configurar o modelo, foi

utilizado o otimizador Adam que faz parte da biblioteca Keras e o erro quadrático médio (*Mean Squared Error* - MSE) para calcular a soma da diferença quadrática entre as imagens. Em seguida, é realizado o treinamento do auto codificador utilizando os dados adquiridos anteriormente, como mostra o Código 4.6.

Código 4.6: Compilando e treinando autocodificador.

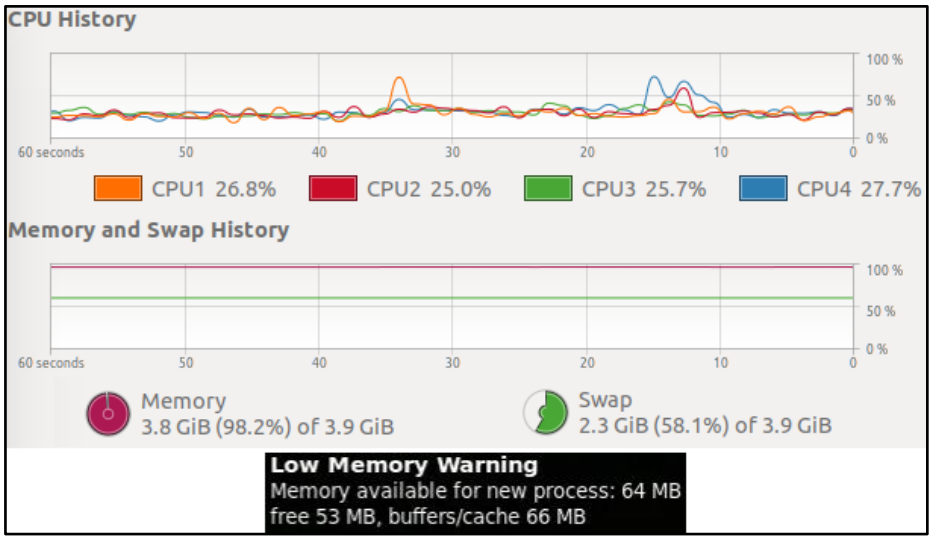
```
autoencoder.compile(optimizer='Adam', loss="mse",metrics=['accuracy'])

history = autoencoder.fit(
    train_images,
    train_images,
    batch_size=Train_Batch_Size,
    epochs=Num_Epochs,
    validation_split=Valid_Split)
```

Fonte: Elaborada pelo autor.

Os parâmetros utilizados na função *fit* (treinamento) são as imagens de treino, para *Num_Epochs* = 15, representando em quantas épocas será realizado o treinamento e *validation_split* = 0.2, referindo que 20% das imagens utilizadas para treinamento serão utilizadas para validar o mesmo. Dessa forma, como o lote de treinamento possui 7000 imagens, 5600 serão utilizadas para treinar e as 1400 imagens restantes para validação.

Figura 14: Monitoramento de CPU, memória principal e memória *swap.*

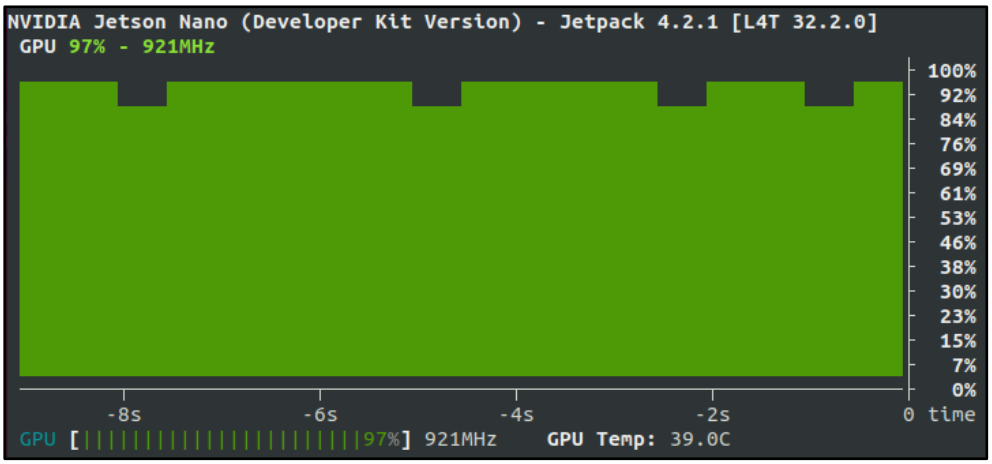


Fonte: Elaborada pelo autor.

Como pode ser observado na Figura 14, durante o processo de treinamento do autocodificador, a CPU tem uso moderado, pois quem está realizando o trabalho

pesado é a GPU. A Figura 15 ilustra o processamento da GPU no momento do treinamento.

Figura 15: Processamento GPU durante processo de treinamento.



Fonte: Elaborada pelo autor.

Já o uso da memória principal chega muito próximo ao seu limite, necessitando constantemente realizar *swap* de memória para se manter executando sem problemas. Como mencionado na seção 4.2.2, a necessidade de aumentar a memória virtual do módulo Jetson Nano de 2 GB para 4 GB fica evidente ao visualizar a utilização em pelo menos 2.3 GB de *swap*.

É possível perceber, em alguns momentos, avisos de pouca disponibilidade de memória para processamento sendo mostrados pelo sistema operacional, mas que não causam um travamento completo do treinamento.

Figura 16: Status de cada época do treinamento do autocodificador utilizando GPU.

```
Epoch 3/15
5600/5600 - 87s 16ms/step - loss: 0.0015 - accuracy: 0.8195 - val_loss: 0.0013 - val_accuracy: 0.8278
Epoch 4/15
5600/5600 - 87s 15ms/step - loss: 0.0013 - accuracy: 0.8240 - val_loss: 0.0012 - val_accuracy: 0.8262
Epoch 5/15
5600/5600 - 87s 15ms/step - loss: 0.0011 - accuracy: 0.8233 - val_loss: 0.0011 - val_accuracy: 0.8095
Epoch 6/15
5600/5600 - 87s 16ms/step - loss: 0.0010 - accuracy: 0.8052 - val_loss: 9.6659e-04 - val_accuracy: 0.8150
Epoch 7/15
5600/5600 - 85s 15ms/step - loss: 9.3483e-04 - accuracy: 0.8211 - val_loss: 9.9452e-04 - val_accuracy: 0.8490
Epoch 8/15
5600/5600 - 86s 15ms/step - loss: 8.7054e-04 - accuracy: 0.8410 - val_loss: 8.0687e-04 - val_accuracy: 0.8509
Epoch 9/15
5600/5600 - 86s 15ms/step - loss: 7.7978e-04 - accuracy: 0.8563 - val_loss: 7.5812e-04 - val_accuracy: 0.8683
```

Fonte: Elaborada pelo autor.

Como o treinamento de um autocodificador é considerado um processo estocástico, isto é, o seu estado é indeterminado com origem em eventos aleatórios,

o resultado deste treinamento é inesperado, podendo convergir para uma acurácia alta e ser um considerado um bom modelo ou ficar preso em uma acurácia baixa e ser considerado um modelo ruim para o sistema de detecção.

Na Figura 16, é possível perceber que entre as épocas 3 e 5 da validação do treinamento (*val_accuracy*) exibido houve uma queda na acurácia do treinamento de 82.78% para 80.95%, mas que voltou a aumentar entre as épocas 6 e 9, onde aumentou de 81.50% para 86.83%.

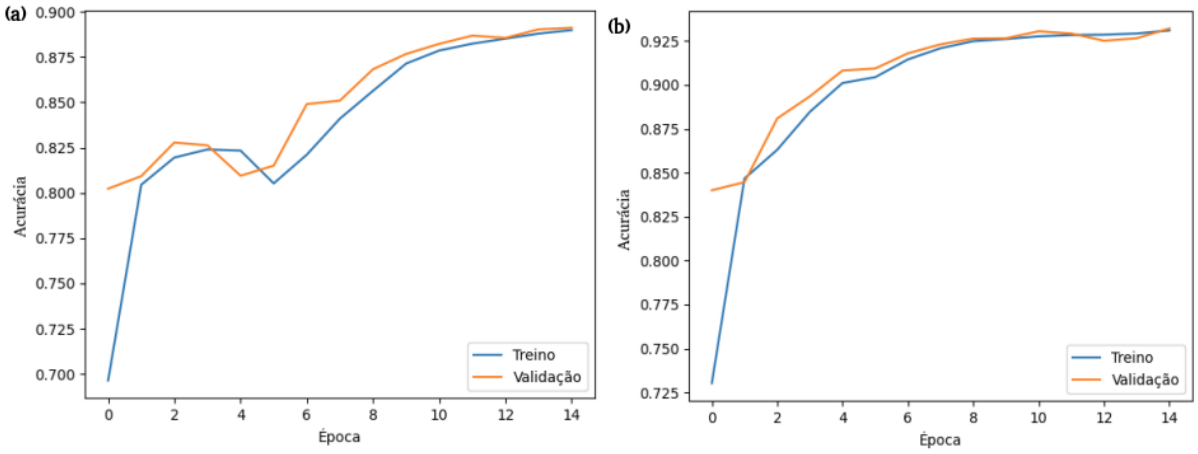
Considerando esses fatos, para chegar a um modelo que tenha características ótimas para o sistema de detecção de anomalias, pode ser necessário fazer diversos treinamentos e compará-los até que o modelo atinja uma acurácia satisfatória.

4.3 Resultados

Os melhores modelos obtidos tiveram uma acurácia entre 90% e 95% na classificação de anomalias, sendo considerados bem satisfatórios para provar o conceito do sistema de detecção de anomalias em sistemas embarcados.

Na Figura 17, é possível observar a evolução da acurácia dos modelos durante as 15 épocas de treinamento. O comportamento do gráfico (a) reafirma o processo estocástico comentado na seção 4.2.3, enquanto o gráfico (b) mostra uma evolução mais progressiva do treinamento. Para os dois modelos, ao final do treinamento, foi-se obtido uma acurácia satisfatória superior a 90%.

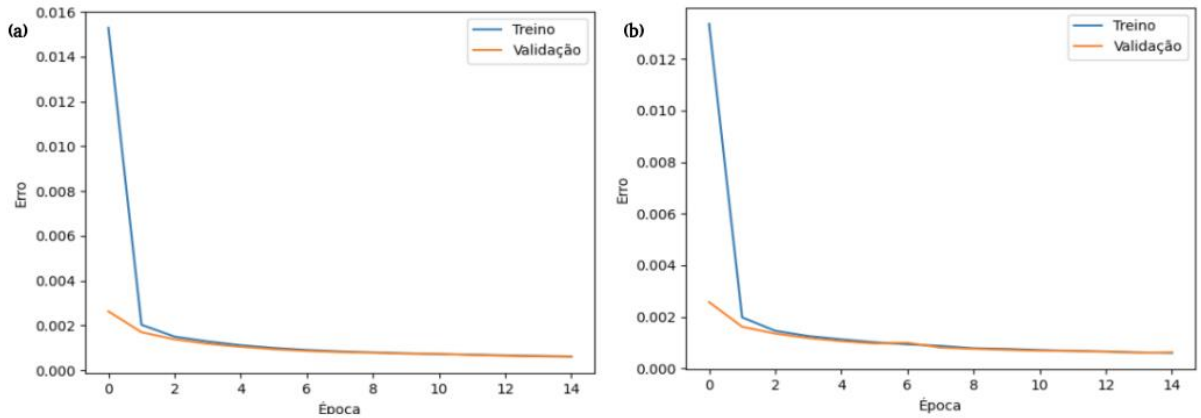
Figura 17: Acurácia de dois modelos satisfatórios (a) e (b).



Fonte: Elaborada pelo autor.

Apesar da diferença nos gráficos de acurácia entre os dois modelos, percebe-se que para os gráficos que exibem o erro do modelo são muito semelhantes, como pode ser observado na Figura 18.

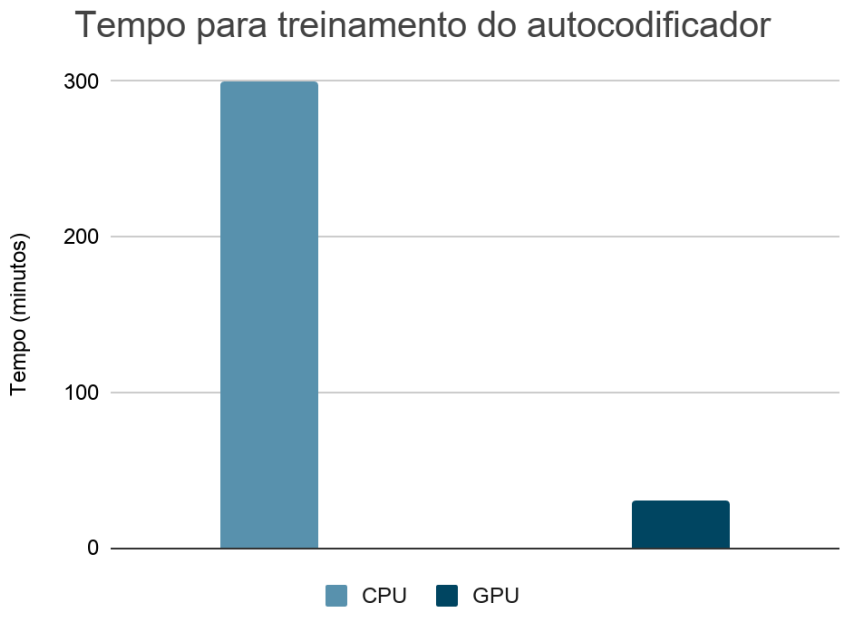
Figura 18: Erro de dois modelos satisfatórios (a) e (b).



Fonte: Elaborada pelo autor.

Em relação à comparação de desempenho entre o treinamento realizado utilizando um processador comum (CPU) e o processador gráfico (GPU) do módulo Jetson Nano, foi possível perceber um ganho de processamento muito alto para o caso da GPU.

Figura 19: Desempenho de treinamento (CPU x GPU).



Fonte: Elaborada pelo autor.

A Figura 19 mostra o tempo, em minutos, levado para a conclusão do treinamento de todas as 15 épocas dos modelos. Para o caso da CPU, totalizando 300 minutos, a média de tempo gasto em cada época foi de cerca de 20 minutos. Já para o caso da GPU, que totaliza 30 minutos, a média de tempo gasto em cada época foi de 1 minuto e 30 segundos.

Considerando esses fatos, o tempo de processamento reduziu de 300 minutos para 30 minutos, ou seja, a GPU só precisa de 10% do tempo gasto por uma CPU para realizar o treinamento de um modelo.

Dessa forma, a utilização de processamento gráfico no sistema de detecção de anomalias pode ser considerado um ponto crucial para garantir a viabilidade de implementação do sistema num processo produtivo da indústria 4.0.

5 Considerações Finais

Este capítulo apresenta as conclusões acerca dos resultados obtidos, bem como possíveis trabalhos futuros para melhoria de algumas características observadas.

5.1 Conclusões

Neste trabalho de conclusão de curso foi estudado um sistema de detecção de anomalias em sistemas embarcados utilizando o poder da técnica dos autocodificadores. O desenvolvimento e a validação desse sistema possibilitam a inserção de novos métodos de testagem não-invasiva ao mercado de sistemas embarcados, garantindo maior controle de qualidade dos produtos manufaturados, buscando, ao mesmo tempo, afetar minimamente o processo produtivo do fabricante das placas.

Além disso, o fato do sistema permitir que o modelo resultante de um treinamento também possa ser reutilizado para classificação das anomalias em outro sistema embarcado de mesmo modelo, torna o sistema ainda mais adaptável e viável em um cenário real.

Os objetivos descritos no Capítulo 1 foram atingidos e se apresentou uma solução candidata ao problema exposto anteriormente, mas apesar dos bons resultados obtidos com o desenvolvimento deste estudo, algumas limitações estão presentes. Dentre elas, destacam-se:

1. Falta de memória principal: O pouco tempo disponível para a confecção do trabalho limitou a solução para uma alternativa que utiliza memória virtual para possibilitar a prova do conceito, mas que não pode ser considerada como o ambiente ideal para desenvolvimento dos modelos.

2. Poucas camadas de codificação e decodificação: Como o módulo Jetson Nano possuía limitações relacionadas à memória principal, poucas camadas de

codificação e decodificação foram utilizadas para validar o modelo. A utilização de mais camadas e/ou de outras camadas que possibilitem a extração de mais características dos dados de entrada por parte do autocodificador podem garantir uma acurácia ainda maior ao modelo.

3. Pré-processamento custoso: Apesar de possibilitar a prova de conceito exposta neste trabalho, o pré-processamento utilizado demanda muita memória do módulo Jetson Nano. A identificação de uma maneira de suavizar o custo do pré-processamento pela biblioteca Keras ou de outra biblioteca que consiga realizar aumento de dados pode melhorar consideravelmente a performance do sistema.

5.2 Trabalhos Futuros

Os seguintes trabalhos futuros são pertinentes, vistas as limitações desta pesquisa:

1. Aquisição de dados de outras plataformas compactas: Os dados utilizados para provar a viabilidade deste trabalho foram coletados de uma placa Tiva C. A coleta de dados de outros sistemas embarcados possibilitaria o teste e a validação da abordagem de forma mais abrangente e tornaria o sistema ainda mais atrativo para a manufatura avançada.

2. Implementar uma ferramenta para automatizar a identificação da placa: De forma não invasiva, o sistema poderia, ao identificar uma placa que já possua um modelo pré-configurado, utilizá-lo para realizar a detecção de anomalias, sem necessidade de perder tempo treinando um novo modelo que teria características muito semelhantes ao configurado anteriormente.

3. Divisão da classe anômala em subclasses: O sistema de detecção de anomalias atualmente classifica as entradas em anômalas ou não-anômalas. O ideal seria utilizar as informações extraídas pelo sistema atual para agregar conhecimento técnico e produzir um sistema complementar, utilizando um algoritmo de aprendizado supervisionado, para garantir ao sistema a capacidade identificar a qual subclasse a anomalia que foi detectada faz parte.

4. Automatizar solução das anomalias identificadas: Em conjunto com o sistema complementar mencionado no item 3 desta seção, a capacidade de realizar procedimentos para solucionar a anomalia identificada de forma automatizada seria um diferencial muito importante para atrair investidores e dar mais visibilidade ao sistema como um todo.

Referências

GILCHRIST, ALASDAIR. **Industry 4.0: the industrial internet of things**. Apress, 2016.

M. R. JOHNSON, "**The Increasing Importance of Utilizing Non-intrusive Board Test Technologies for Printed Circuit Board Defect Coverage**", 2018 IEEE AUTOTESTCON, National Harbor, MD, 2018, pp. 1-5, doi: 10.1109/AUTEST.2018.8532499.

G. DE OLIVEIRA, J. PAULO. **Non-invasive embedded system hardware/firmware anomaly detection based on current signature.** Advanced Engineering Informatics. Submitted September 2020, Under Review.

CHOLLET, FRANÇOIS. **Building Autoencoders in Keras.** Online 2016. https://blog.keras.io/building-autoencoders-in-keras.html - último acesso: 17/03/2021.

ANTONIO L. ALFEO, MARIO G.C.A. CIMINO, GIUSEPPE MANCO, ETTORE RITACCO, GIGLIOLA VAGLINI. **Using an autoencoder in the design of an anomaly detector for smart manufacturing**. Pattern Recognition Letters 136 (2020) 272–278.b.

ABDUL MAJEEB, WENTING DAI, MARIUS ERDT, ALEXEI SOURIN. **One class based feature learning approach for defect detection using deep autoencoders.** Advanced Engineering Informatics. Volume 42, October 2019, 100933. https://doi.org/10.1016/j.aei.2019.100933.

COOK, S. **CUDA Programming: A Developer's Guide to Parallel Computing with GPUs (1st ed.).** San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. 2012.

NVIDIA **Jetson Nano Developer Kit.** 2019. https://developer.nvidia.com/embedded/jetson-nano-developer-kit - último acesso: 25/04/2021.

MCCULLOCH, W.S., PITTS, W. **A logical calculus of the ideas immanent in nervous activity.** *Bulletin of Mathematical Biophysics* 5, 115-133 (1943). https://doi.org/10.1007/BF02478259.

HAYKIN, SIMON S. **Redes Neurais - 2ed.** Bookman, 2001.

HAYKIN, SIMON S. **Neural Networks: A Comprehensive Foundation.** Prentice-Hall, Englewood Cliffs. 1998.

PEARSON, KARL, 1901. **On lines and planes of closest fit to systems of points in space,** *Philosophical Magazine,* Series 6, vol.2, no. 11, pp. 559-572.

LOWE, D.G. **Distinctive Image Features from Scale-Invariant Keypoints.** International Journal of Computer Vision 60, 91-110 (2004). https://doi.org/10.1023/B:VISI.0000029664.99615.94.

T.I. **TM4C LaunchPad Evaluation Kit.** 2013. https://www.ti.com/lit/wp/spmy010/spmy010.pdf?ts=1619490202644 - último acesso: 25/04/2021.

CHOLLET, FRANÇOIS et al. **Keras.** 2015. https://keras.io. Github - https://github.com/fchollet/keras.

MARTÍN ABADI, ASHISH AGARWAL, PAUL BARHAM et al. **TensorFlow: Large-scale machine learning on heterogeneous systems**, 2015. Software available from tensorflow.org.

BRADSKI G. **The OpenCV Library.** Dr Dobb's Journal of Software Tools. 2000.

Jetson Nano, **Jetson Nano Module Datasheet.** 2019. https://static6.arrow.com/aropdfconversion/1280937d98cee0191d01d75dc0cb547310d0fa57/jetson-nano-module-datasheet-us-1031771-r3-hr.pdf - último acesso: 25/04/2021.