INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE SANTA CATARINA - CÂMPUS FLORIANÓPOLIS
DEPARTAMENTO ACADÊMICO DE METAL MECÂNICA
CURSO DE GRADUAÇÂO EM ENGENHARIA MECATRÔNICA

ANDRÉ ALEXANDRE MÜLLER

ESTUDO DO FRAMEWORK TENSORFLOW LITE PARA USO EM PROJETOS MECATRÔNICOS MICROCONTROLADOS

FLORIANÓPOLIS, 2023.

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE SANTA CATARINA - CÂMPUS FLORIANÓPOLIS
DEPARTAMENTO ACADÊMICO DE METAL MECÂNICA
CURSO DE GRADUAÇÂO EM ENGENHARIA MECATRÔNICA

ANDRÉ ALEXANDRE MÜLLER

ESTUDO DO FRAMEWORK TENSORFLOW LITE PARA USO EM PROJETOS MECATRÔNICOS MICROCONTROLADOS

Trabalho de Conclusão de Curso submetido ao Instituto Federal de Educação, Ciência e Tecnologia de Santa Catarina como parte dos requisitos para obtenção do título de Engenheiro em Mecatrônica.

Orientador:
Prof. Valdir Noll, Dr. Eng.

FLORIANÓPOLIS, 2023.

Ficha de identificação da obra elaborada pelo autor.

Müller, André Alexandre
ESTUDO DO FRAMEWORK TENSORFLOW LITE PARA USO EM PROJETOS **MECATRÔNICOS MICROCONTROLADOS / André Alexandre** Müller; **orientação de Valdir Noll. - Florianópolis,** SC, 2023.
107 p.

Trabalho de Conclusão de Curso (TCC) - Instituto Federal **de Santa Catarina, Câmpus Florianópolis.** Bacharelado **em Engenharia Mecatrônica.** Departamento **Acadêmico de Metal** Mecânica.
Inclui Referências.

1. TensorFlow Lite. 2. Microcontroladores. 3. Sistemas **Embarcados. 4. Aprendizado de Máquina. I.** Noll, **Valdir. II. Instituto Federal de Santa Catarina. III.** ESTUDO **DO FRAMEWORK TENSORFLOW LITE PARA USO EM** PROJETOS **MECATRÔNICOS** MICROCONTROLADOS.

ESTUDO DO FRAMEWORK TENSORFLOW LITE PARA USO EM PROJETOS MECATRÔNICOS MICROCONTROLADOS

ANDRÉ ALEXANDRE MÜLLER

Este trabalho foi julgado adequado para obtenção do título de Engenheiro em Mecatrônica e aprovado na sua forma final pela banca examinadora do Curso Engenharia Mecatrônica do Instituto Federal de Educação, Ciência e Tecnologia de Santa Catarina.

Florianópolis, 9 de fevereiro, 2023.

Banca Examinadora:

IFSC
Documento assinado digitalmente
VALDIR NOLL
Data: 10/02/2023 15:45:08-0300
CPF: 425.839.000-30
Verifique as assinaturas em https://v.ifsc.edu.br

Valdir Noll, Prof. Dr. Eng.
Instituto Federal de Santa Catarina

IFSC
Documento assinado digitalmente
MARIO DE NORONHA NETO
Data: 13/02/2023 09:31:52-0300
CPF: 003.859.519-22
Verifique as assinaturas em https://v.ifsc.edu.br

Mario de Noronha Neto, Prof. Dr. Eng.
Instituto Federal de Santa Catarina

gov.br
Documento assinado digitalmente
FRANCISCO EDSON NOGUEIRA DE MELO
Data: 10/02/2023 16:00:41-0300
Verifique em https://verificador.iti.br

Francisco Edson Nogueira de Melo, Prof. Dr. Eng.
Instituto Federal de Santa Catarina

MINISTÉRIO DA EDUCAÇÃO
SECRETARIA DE EDUCAÇÃO PROFISSIONAL E TECNOLÓGICA
INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE SANTA CATARINA
CAMPUS FLORIANÓPOLIS

DECLARAÇÃO DE FINALIZAÇÃO DE TRABALHO DE CURSO

Declaro que o estudante ANDRE ALEXANDRE MULLER, matrícula nº 1520061218, do Curso de Engenharia Mecatrônica, defendeu o trabalho intitulado ***ESTUDO DO FRAMEWORK TENSORFLOW LITE PARA USO EM PROJETOS MECATRONICOS MICROCONTROLADOS***, o qual está apto a fazer parte do banco de dados da Biblioteca Hercílio Luz do Instituto Federal de Santa Catarina, Campus Florianópolis.

Florianópolis, 9 de fevereiro de 2023.

Documento assinado digitalmente
VALDIR NOLL
Data: 10/02/2023 15:46:15-0300
CPF: 425.839.000-30
Verifique as assinaturas em https://v.ifsc.edu.br

Prof. Orientador do TCC: Valdir Noll

Dedico este trabalho a Deus; o criador,
Aos meus pais; pelo apoio incondicional,
À minha esposa e filha; por todo o seu amor.

AGRADECIMENTOS

Agradeço primeiramente aos meus pais, Mario Roberto Müller e Jacira Silveira dos Passos Müller, por toda a sua dedicação e apoio ao longo da vida, inclusive durante a minha trajetória acadêmica e profissional. Vocês são muito especiais e certamente são a razão para tudo o que foi possível conquistar até aqui.

Também agradeço à minha esposa e à minha filha por todo o seu amor. Saibam que vocês me motivam a ser uma pessoa melhor e à buscar novas conquistas. Eu sou uma pessoa muito melhor com vocês por perto.

Ao meu orientador Valdir Noll, que além de me apoiar ao longo de toda a minha trajetória acadêmica, algo que intrinsecamente já possui um imenso valor, ainda é um amigo para todas as horas.

Não poderia deixar de resaltar e agradecer às outras pessoas que me ajudaram de uma forma ou de outra ao longo deste trabalho. Seja por uma idéia, uma conversa, um direcionamento ou até mesmo por disponibilizar algum conteúdo de qualidade, vocês certamente me ajudaram a ter uma melhor compreensão do assunto aqui abordado.

Aqui incluo em especial o pesquisador Peter Warden, que mesmo não o conhecendo pessoalmente, foi imprescindivel para podermos compreender o conteúdo de aprendizado de máquina com TensorFlow Lite, transformando nosso entendimento sobre o mesmo neste trabalho. O seu esforço e dedicação no ensinamento dessa tecnologia é excelente, e fundamental para aqueles que buscam sua iniciação no tema.

Por fim, porém não menos importante, gostaria de agradecer aos meus amigos, e aqui incluo o meu irmão, pela parceria e apoio de sempre. Sei que já usamos muitas horas falando dos mais diversos assuntos, e certamente não foi diferente com o tema abordado neste trabalho. Obrigado por serem sempre ouvidos e por se fazerem escutar quando preciso.

"The true sign of intelligence is not knowledge but imagination."
Albert Einstein

RESUMO

O desenvolvimento de software e sistemas inteligentes tem relevante importância para a geração de aplicações modernas na área da mecatrônica. Ele desempenha um papel crucial para o valor agregado de produtos e processos na área em comparação com aqueles cuja abordagem de projeto, dito como clássico, não involve sistemas e computação avançada. Atualmente, o aprendizado de máquina é um dos ramos mais comentados e explorados na área da computação e começa a despertar interesse nos diversos ramos da engenharia, apresentando diversos benefícios e capacidades únicas, além de muitas vezes interagir de perto com a teoria de controle, algo muito presente nas aplicações mecatrônicas. O objetivo principal deste trabalho é estudar o framework TensorFlow Lite, buscando entender a sua complexidade e aplicabilidade no que tange o desenvolvimento de aplicações mecatrônicas. Estudamos os exemplos de referência ofertados pelo framework com o objetivo de entender se eles podem ser utilizados como solução para algumas funcionalidades de natureza mecatrônica. O trabalho também avalia a utilidade do framework em aplicações mecatrônicas microcontroladas de borda, situação onde entendesse que o mesmo possui um grau de importância maior. O estudo apresentado contou com uma abordagem qualitativa, onde correlacionou a aplicabilidade e adaptabilidade dos exemplos de referência em outros contextos tais como um projeto mecatrônico. Apesar dos exemplos de referência serem muito úteis e adaptáveis, o desenvolvimento de aplicações avançadas ainda carece de um profissional com conhecimento teórico suficiente, ou mesmo avançado, que consiga modelar diferentes objetivos. Apesar de cobrir uma boa gama de possíveis aplicações mecatrônicas, essa necessidade torna difícil a aplicação do TensorFlow Lite como solução universal para toda e qualquer funcionalidade presente em uma aplicação mecatrônica.

Palavras-chave: TensorFlow Lite, Microcontroladores, Sistemas Embarcados, Aprendizado de Máquina.

ABSTRACT

Developing software and intelligent systems is essential for developing modern mechatronics applications. Systems and advanced computing play a crucial role in enhancing the added value of products and processes in the area, compared to classic design approaches. Currently, machine learning is one of the most commented on and explored branches in the area of computing and beginning to arouse interest in various fields of engineering, presenting several unique benefits and capabilities, as well as often intertwining with control theory, which is very prevalent in mechatronic applications. It is the main objective of this work to study the TensorFlow Lite framework, in order to gain a better understanding of its complexity and applicability in the development of mechatronic applications. Studying the reference examples offered by the framework allowed us to determine whether they could be used to solve some mechatronic problems. It also evaluates the framework's usefulness in microcontrolled mechatronic edge applications, where its value is greater. Based on a qualitative approach, the present study correlated the applicability of reference examples in other contexts, such as mechatronics. In spite of the usefulness and adaptability of the reference examples, the development of advanced applications still lacks a professional with sufficient theoretical knowledge, or even someone who is advanced enough to be able to model different goals. This needs makes it difficult to apply TensorFlow Lite as a universal solution to any and all functionality in a mechatronic application, despite its good coverage of mechatronic applications.

Keywords: TensorFlow Lite. Microcontrollers. Embedded Systems. Machine Learning.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
ARM	Fabricante de dispositivos eletrônicos
Bluetooth-LE	Tecnologia de comunicação sem fio de baixo consumo
C++	Linguagem de programação
CPU	Unidade de processamento central
Cortex-M	Núcleo de microcontrolador
Deep Learning	Ramo do aprendizado de máquina que tenta modelar abstrações de alto nível de dados
IDE	Ambiente de Desenvolvimento Integrado
IoT	Internet das Coisas
Python	Linguagem de programação
RAM	Memória de acesso aleatório

SUMÁRIO

1 INTRODUÇÃO

A mecatrônica é uma área do conhecimento de característica multidisciplinar que inclui partes da engenharia mecânica, eletrônica e da computação. Dentre suas aplicações típicas estão: a automação industrial, a robótica e a usinagem. A teoria de controle de processos é um tópico cuja importância leva a ser um dos requisitos para a formação de pessoal nestas áreas. Recentemente, tópicos que envolvem inteligência artificial e aprendizado de máquina vem se tornando muito importantes para o desenvolvimento de aplicações modernas, porém, ainda são pouco conhecidas nos cursos de mecatrônica.

Algumas técnicas de aprendizado de máquina já estão integradas em aplicações mecatrônicas modernas e utilizam componentes comerciais. Dentre os exemplos, podemos destacar os sensores para reconhecimento de padrões de voz e imagem. Estes componentes são usualmente adquiridos ou licenciados e por fim integrados às aplicações.

Estas aplicações, a robótica, a fabricação flexível, os veículos autônomos e a logística, para mencionar os principais exemplos, exigirão dos profissionais de engenharia que entendam suficientemente do arcabouço técnico no qual essas aplicações foram projetadas, de forma que possam otimizar seu funcionamento. Isso inclui, portanto, os algoritmos de inteligência artificial e o aprendizado de máquina.

Essa demanda dá origem à necessidade de se compreender tópicos de inteligência artificial e que possibilite o desenvolvimento de sistemas mecatrônicos mais avançados. Este trabalho, como forma de contribuição para o contexto apresentado, pretende de forma técnica e experimental, avaliar o uso do TensorFlowLite para projetos de sistemas mecatrônicos que envolvem microcontroladores.

Esta pesquisa não pretende cobrir totalmente as possíveis técnicas e teorias que envolvem os tópicos de inteligência artificial e aprendizado de máquina, porém apresenta diversos casos de uso que podem ser readequados para utilização futura, possibilitando sua compreensão e aplicação prática.

1.1 Justificativa

O software agregado a sistemas inteligentes tem um impacto significativo nas aplicações mecatrônicas modernas, desempenhando um papel fundamental no que diz respeito ao valor agregado destas soluções, se comparadas às abordagens clássicas.

Embora os tópicos da inteligência artificial e aprendizado de máquina sejam relativamente comuns em currículos como o da ciência da computação, há necessidade de incorporação destes temas também em algumas engenharias, principalmente a a

engenharia mecatrônica. Estudar, de forma exploratória, uma das tecnologias mais promissoras relativas à inteligência artificial, assim como identificar os possíveis benefícios quando esta tecnologia é aplicada a projetos mecatrônicos microcontrolados é uma das formas de trazer à tona informações e ampliar o debate sobre este importante assunto.

Este trabalho busca trazer uma abordagem colaborativa no que concerne às novas aplicações neste espaço, de sorte que possam emergir com mais segurança. Não há ainda, conforme explicitado pelos desenvolvedores do TensorFlow Lite (ou TinyML), uma aplicação comercial amplamente conhecida, entretanto sabe-se que existem diversas aplicações que poderiam se beneficiar dela. Como mencionado pela equipe do TinyML, espera-se que especialistas que trabalham nos domínios tais como: agricultura, exploração espacial, medicina, bens consumíveis e a mecatrônica possam se beneficiar e resolver os problemas dessas áreas usando ferramentas, tais como o TinyML, por conta própria, ou ao menos conseguir comunicar que tipos de problemas essas áreas possuem, para que então outros especialistas e pesquisadores possam contribuir.

1.2 Definição do Problema

Entender o funcionamento da tecnologia TensorFlow e identificar os benefícios que esta ferramenta possa trazer, quando aplicada a projetos mecatrônicos microcontrolados.

1.3 Objetivo Geral

Estudar e avaliar a utilidade do Framework TensorFlow Lite para aplicações mecatrônicas microcontroladas.

1.4 Objetivos Específicos

Os objetivos específicos deste trabalho são:

a) Estudar o processo de desenvolvimento de aplicações microcontroladas inteligentes utilizando o Framework TensorFlow Lite;

b) Selecionar e obter as plataformas de desenvolvimento microcontroladas que serão utilizadas neste e em futuros trabalhos;

c) Documentar o processo de desenvolvimento apresentando exemplos e casos-de-uso;

d) Discutir as vantagens, desvantagens e limitações do Framework TensorFlow Lite;

e) Sugerir possíveis aplicações mecatrônicas que possam se beneficiar do uso de inteligência artificial.

1.5 Estrutura do Trabalho

Este trabalho está dividido em três partes: a fundamentação teórica, que foi adicionada como um anexo por ser talvez não completa o suficiente, a apresentação dos resultados e os apêndices, que contém o desenvolvimento técnico dos casos de uso.

Na fundamentação teórica apresentamos o framework TensorFlow, que é utilizado para implementação prática dos conceitos de Aprendizado de Máquina (Machine Learning). O anexo C faz uma revisão integrativa dos conceitos de inteligência artificial, mais especificamente no Aprendizado de Máquina.

Aborda-se desde o treinamento de uma rede neural até a geração do código binário a ser embarcado nos microcontroladores. Estes exemplos são importantes, pois podem ser utilizados como ponto de partida para o desenvolvimento de aplicações mais complexas e de aplicação comercial

Na discussão dos resultados, apresentamos as vantagens e desvantagens do framework TensorFlow, obtidos de forma experimental, assim como exemplos de projetos no ramo da mecatrônica que poderiam se beneficiar desta ferramenta.

2 FUNDAMENTAÇÃO TEÓRICA

O objetivo deste capítulo é apresentar a Inteligência Artificial, especificamente o Aprendizado de Máquina, de forma simplificada, com foco prático, e mostrar como esses conceitos podem ser implementados com o framework TensorFlow Lite em microcontroladores.

2.1 O que é TensorFlow?

O TensorFlow é uma plataforma de código aberto que engloba todas as etapas de desenvolvimento de aplicações que utilizam o aprendizado de máquina (TENSORFLOW, 2022f). Ele possui um arcabouço abrangente e flexível de ferramentas, bibliotecas e comunidades que permitem que os pesquisadores atuem no estado da arte do aprendizado de máquina. Ele ainda permite que desenvolvedores criem e implementem facilmente aplicativos com a tecnologia de aprendizado de máquina.

O TensorFlow foi originalmente desenvolvido por pesquisadores e engenheiros que trabalham na equipe do Google Brain dentro da área de pesquisa de inteligência de máquina do Google, cujo objetivo é a pesquisa com foco no aprendizado de máquina e redes neurais profundas. O sistema é bastante flexível e pode ser usado em uma ampla variedade de aplicações.

O TensorFlow fornece API's para Python e C++ estáveis, bem como APIs compatíveis para outras linguagens (TENSORFLOW, 2022a, p. 20).

2.1.1 TensorFlow Lite para Microcontroladores

O TensorFlow Lite for para Microcontroladores (TFLM) foi projetado para permitir a execução de modelos de aprendizado de máquina em microcontroladores e outros dispositivos que possuam restrições de capacidade de memória (TENSORFLOW, 2022g).

Por exemplo, o runtime do núcleo do TFLM usa apenas 16 KB em um Arm Cortex-M3 e pode executar muitos modelos básicos de aprendizado. Ele não requer suporte ao sistema operacional, nenhuma biblioteca padrão C ou C++ ou alocação dinâmica de memória.

O TFLM foi projetado como um *framework* que permite executar, em microcontroladores, um subconjunto de modelos TensorFlow Machine Learning / Deep Learning. O TFLM permite que os engenheiros da área de aprendizado de máquina contornem os desafios de portar estes modelos para dispositivos embarcados, oferecendo uma biblioteca C++ para interpretação em tempo de execução e inferência de modelos (MEDIUM, 2021).

2.1.2 Plataformas de desenvolvimento suportadas

O TensorFlow Lite para microcontroladores é escrito em C++ 11 e requer uma plataforma de 32 bits. Ele foi testado extensivamente com muitos processadores baseados na arquitetura ARM Cortex-M Series (ARM, 2022) e foi também portado para outras arquiteturas, incluindo o ESP32 (ESPRESSIF, 2022).

O framework também está disponível como uma biblioteca Arduino, assim como também permite a geração de projetos para outros ambientes de desenvolvimento. Trata-se de uma plataforma de código aberto e pode ser incluído em qualquer projeto C++ 11.

Algumas das placas de desenvolvimento suportadas pelo TensorFlow Lite são apresentadas na tabela 3.

Tabela 1 – Plataformas de desenvolvimento com suporte

Plataforma	CPU Flash	SRAM	Núcleo	Preço
Arduino Nano 33 BLE Sense	1MB	256KB	Cortex M4 32-bit, 64MHz	U$ 43,50
SparkFun Edge	1MB	348KB	Cortex-M4F 32-bit, 48MHz	U$ 16,50
STM32F746 Discovery Kit	1MB	340KB	Cortex-M7 32-bit, 216 MHz	U$ 55,12
Adafruit EdgeBadge	512KB	192KB	Cortex-M4F 32-bit 120MHz	U$ 35,95
Adafruit TensorFlow Lite for Microcontrollers Kit	512KB	192KB	Cortex-M4F 32-bit, 120MHz	U$ 44,95

Fonte: Do autor.

2.2 O processo de desenvolvimento com o TensorFlow Lite

As etapas a seguir são necessárias para implantar e executar um modelo do TensorFlow em um microcontrolador:

2.2.1 Treinar um modelo

Gerar um modelo pequeno do TensorFlow, que contenha operações compatíveis e que se ajuste ao dispositivo escolhido (TENSORFLOW, 2022e).

Converter para um modelo do TensorFlow Lite usando o conversor do TensorFlow Lite (TENSORFLOW, 2022b).

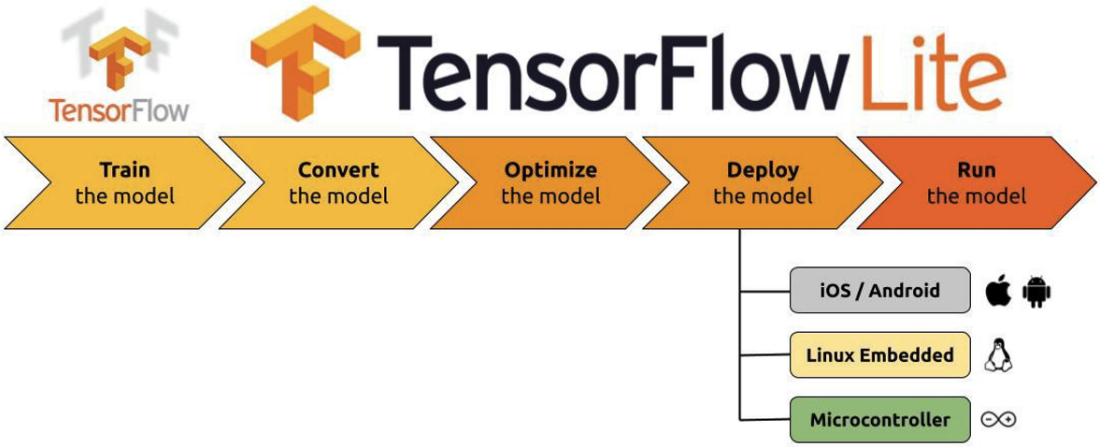
2.2.2 Otimizar e implantar

Converter para uma matriz de bytes no padrão C usando ferramentas padrão para armazená-lo em uma memória de programa, do tipo somente leitura, no dispositivo microcontrolado (TENSORFLOW, 2022c).

2.2.3 Executar a inferência

Processar o modelo no dispositivo usando a biblioteca C++. A figura 9 representa o fluxo de trabalho de implantação de um modelo do TensorFlow usando TFLM. Observe que o dispositivo de destino pode ser qualquer sistema que suporte C++ e tenha os requisitos mínimos de processamento, memória e sistema operacional. Muitos dos exemplos que estaremos abordadando no capítulo 4 usam apenas um *super-loop*, não necessitando de um sistema operacional.

Figura 1 – Fluxo de desenvolvimento do TensorFlow Lite



Fonte: MEDIUM, 2021

2.3 Arquitetura dos modelos

Ao projetar um modelo para uso em microcontroladores, é importante considerar o tamanho do modelo, a carga de trabalho e as operações usadas (TENSORFLOW, 2022c).

2.3.1 Tamanho do modelo

Um modelo deve ser pequeno o suficiente para caber na memória do dispositivo escolhido junto com o seu programa, considerando, principalmente a memória ocupada em tempo de execução.

Para criar modelos menores, pode-se usar camadas cada vez menores em sua arquitetura. No entanto, modelos pequenos são mais propensos a sofrer de *sub-ajuste*. Isso significa que para muitos problemas, faz sentido tentar usar o maior

modelo, desde que a memória do dispositivo escolhido suporte. O programa principal do TFLM utiliza aproximadamente 16 KB em um Cortex-M3.

2.3.2 Carga de trabalho

O tamanho e a complexidade do modelo têm impacto na carga de trabalho. Modelos grandes e complexos podem resultar em um ciclo de trabalho mais alto, o que significa que o processador do dispositivo usado gastará mais tempo trabalhando e menos tempo livre para executar outras tarefas. Isso aumentará o consumo de energia e a dissipação de calor, o que pode ser um problema dependendo da aplicação.

2.3.3 Suporte à operação

Atualmente, o TFLM oferece suporte a um subconjunto limitado de operações do TensorFlow, o que afeta as arquiteturas de modelo que podem ser executadas. Há em andamento um trabalho no sentido da ampliação do suporte à operação, tanto em implementações de referência quanto em otimizações para arquiteturas específicas.

2.4 A biblioteca do TensorFlow Lite

A biblioteca C++ do TensorFlow Lite for Microcontrollers faz parte do repositório do TensorFlow. Ela foi projetada para ser legível, fácil de modificar, bem testada, fácil de integrar e compatível com o TensorFlow Lite comum.

O documento a seguir descreve a estrutura básica da biblioteca C++ e fornece informações sobre como criar o seu próprio projeto.

2.4.1 Estrutura de arquivos

O diretório raiz do TFLM é o */micro* e ele possui uma estrutura relativamente simples. No entanto, como ele está localizado dentro do extenso repositório do TensorFlow. Foram criados scripts e arquivos de projeto pré-gerados que fornecem os arquivos de origem relevantes isoladamente em várias IDE's.

2.4.2 Arquivos e diretórios principais

Os arquivos mais importantes para usar o interpretador TFLM estão na raiz do projeto, acompanhados de testes:

a) O *all_ops_resolver.h* ou *micro_mutable_op_resolver.h* podem ser usados para fornecer as operações usadas pelo interpretador para executar o modelo. Como o *all_ops_resolver.h* puxa todas as operações disponíveis, ele usa muita memória. Em aplicativos em produção, deve-se usar o *micro_mutable_op_resolver.h* para extrair apenas as operações que o modelo a ser utilizado precisa.

b) O *micro_error_reporter.h* gera informações para depuração.

c) E o *micro_interpreter.h* contém código para manipular e executar modelos.

O sistema de compilação fornece implementações específicas de arquivos para diferentes plataformas de desenvolvimento. Eles estão localizados em um diretório com o nome da plataforma, por exemplo, *SparkfunEdge*.

Existem vários outros diretórios, incluindo:

a) O */kernel*, que contém implementações de operação e o código associado.

b) O */tools*, que contém ferramentas de construção e suas saídas.

c) E o */examples*, que contém códigos exemplo.

2.4.3 Como iniciar um novo projeto

O TFLM é capaz de gerar projetos para diversas plataformas usando um Makefile. Os ambientes atualmente suportados são Keil, Make e Mbed.

Para gerar esses projetos com o Make, basta clonar o repositório do TensorFlow e executar o seguinte comando:

```
make -f tensorflow/lite/micro/tools/make/Makefile generate_projects
```

Isso levará alguns minutos, pois precisa diversas dependências. Quando terminado, é possível visualizar algumas pastas criadas dentro de um diretório como o *gen/linux_x86_64/prj/* (o caminho exato depende do sistema operacional onde foi executado o comando make). Essas pastas contêm o projeto gerado e os arquivos de origem.

Depois de executar o comando, pode-se encontrar os projetos *hello_world* em *gen/linux_x86_64/prj/hello_world*. Por exemplo, *hello_world/keil* conterá o projeto Keil.

2.5 Executando os testes unitários e construindo os binários

Para construir a biblioteca e executar todos os seus testes unitários deve-se executar o comando:

```
make -f tensorflow/lite/micro/tools/make/Makefile test
```

Para executar um teste individual, use o seguinte comando, substituindo *<test_name>* pelo nome do teste:

```
make -f tensorflow/lite/micro/tools/make/Makefile test_<test_name>
```

É possível encontrar os nomes dos testes nos Makefiles do projeto. Por exemplo, *examples/hello_world/Makefile.inc* especifica os nomes de testes para o exemplo *hello_world*.

Para construir um binário executável para um determinado projeto (como um aplicativo de exemplo), deve-se usar o seguinte comando, substituindo *<project_name>* pelo projeto que se deseja construir:

```
make -f tensorflow/lite/micro/tools/make/Makefile <project_name>_bin
```

Como exemplo, o comando a seguir criará um binário para o aplicativo *hello_world*:

```
make -f tensorflow/lite/micro/tools/make/Makefile hello_world_bin
```

Por padrão, o projeto será compilado para o sistema operacional onde os comandos *make* estão sendo executados. Para especificar uma arquitetura diferente, deve-se usar o parâmetro *TARGET=*. O exemplo a seguir mostra como criar o exemplo *hello_world* para a plataforma de desenvolvimento SparkFunEdge:

```
make -f tensorflow/lite/micro/tools/make/Makefile \
TARGET=sparkfun_edge hello_world_bin
```

Quando um destino é especificado, quaisquer arquivos de origem específicos de destino disponíveis serão usados no lugar do código original. Por exemplo, o subdiretório *examples/hello_world/sparkfun_edge* contém implementações SparkFunEdge dos arquivos *constants.cc* e *output_handler.cc*, que serão usados quando o destino *sparkfun_edge for especificado*.

É possível encontrar os nomes dos projetos nos Makefiles do projeto. Por exemplo, *examples/hello_world/Makefile.inc* especifica os nomes binários para o exemplo *hello_world*.

2.5.1 Utilizando kernels otimizados

Os kernels de referência na raiz de tensorflow/lite/micro/kernels são implementados em C/C++ puro e não incluem otimizações de hardware específicas da plataforma. Versões otimizadas de kernels são fornecidas em subdiretórios. Por exemplo, *kernels/cmsis-nn* contém vários kernels otimizados que fazem uso da biblioteca *CMSIS-NN* do ARM.

Para gerar projetos usando kernels otimizados, deve-se usar o seguinte comando, substituindo *<subdirectory_name>* pelo nome do subdiretório que contém as otimizações:

```
make -f tensorflow/lite/micro/tools/make/Makefile \
TAGS=<subdirectory_name> generate_projects
```

2.5.2 Gerando uma biblioteca para projetos com Arduino

Se for preciso gerar uma nova compilação da biblioteca, deve-se executar o seguinte script no repositório do TensorFlow:

```
./tensorflow/lite/micro/tools/ci_build/test_arduino.sh
```

Em *gen/arduino_x86_64/prj/tensorflow_lite.zip* pode-se encontrar a biblioteca resultante após a executação do comando anterior.

2.5.3 Portando para outros dispositivos

Orientações sobre a portabilidade do TFLM para novas plataformas e dispositivos podem ser encontradas na documentação oficial (GITHUB, 2021).

2.6 Exemplos de referência e material complementar

Os recursos do TensorFlow Lite for Microcontrollers Experiments (GOOGLE, 2022b) apresentam funcionalidades desenvolvidas por desenvolvedores que combinam Arduino e TensorFlow para criar experiências e ferramentas únicas.

Existem outros recursos de aprendizado e desenvolvimento de código aberto disponíveis. Mais notavelmente os próprios exemplos apresentados no GitHub do TensorFlow, que explicam passo a passo como desenvolver modelos, treinar e implantar aplicativos em placas de desenvolvimento.

Cada aplicativo de exemplo no GitHub tem um arquivo *README.md* que explica como ele pode ser implantado nas plataformas suportadas. Alguns exemplos também possuem tutoriais passo-a-passo usando uma plataforma específica, conforme abaixo:

a) *hello_world*: Demonstra o básico absoluto do uso do TensorFlow Lite para microcontroladores. Este tutorial usa qualquer dispositivo suportado

b) *micro_speech*: Captura áudio com um microfone para detectar as palavras "sim"e "não". Este tutorial usa SparkFun Edge

c) *magic_wand*: Captura dados do acelerômetro para classificar três gestos físicos diferentes. Este tutorial usa Arduino Nano 33 BLE Sense

d) *person_detection*: Captura dados da câmera com um sensor de imagem para detectar a presença ou ausência de uma pessoa

Por fim, há o Tiny Machine Learning Open Education Initiative (TinyMLedu). Trata-se de um grupo internacional que trabalha para melhorar o acesso global a materiais educacionais para o campo de ponta do TinyML.

A implantação bem-sucedida neste campo requer conhecimento de aplicativos, algoritmos, hardware e software. O projeto TinyMLedu é hospedado pela Harvard John A. Paulson School of Engineering and Applied Sciences em colaboração com a TinyML Foundation, e contém vários cursos de educação, livros, slides e material para ajudar as pessoas a adquirir os fundamentos necessários para trabalhar nesta área (TINYMLEDU, 2022).

3 METODOLOGIA

Este trabalho tem como finalidade a pesquisa básica sobre o estado da arte da inteligência artificial e aprendizado de máquina para aplicações microcontroladas em Mecatrônica, com ênfase no TensorFlow Lite. Tem por objetivo aprofundar o conhecimento científico neste tema, pouco abordado ainda no contexto da Engenharia Mecatrônica.

Parte desse objetivo foi alcançado pelo Capitulo 2,que trata extensivamente do tema de Aprendizado de Máquina aplicada a Microcontroladores, que faz então o processamento na borda, ou seja, na ponta da cadeia de aquisição, tratamento e controle de processos.

A classificação da pesquisa quanto aos seus objetivos, se divide em três grandes grupos: exploratórias, descritivas e explicativas (ARAÚJO ANEIDE OLIVEIRA; OLIVEIRA, 1997). A opção que mais se aproximou ao tipo deste trabalho é a exploratória.

O principal objetivo de uma pesquisa exploratória é a obtenção de *insights* e ideias. Muitas vezes, no início de um estudo, os problemas a serem investigados não estão totalmente definidos e faltam informações para a sua compreensão completa. Neste caso, estudaremos o TensorFlow Lite verificando a sua aplicabilidade no contexto dos projetos mecatrônico microcontrolados e sugerindo, a partir daí possíveis aplicações.

Este estudo apresentado conta com a abordagem qualitativa e teve como foco o Estudo dos exemplos de referência do TensorFlow Lite buscando correlacionar sua aplicabilidade em projetos de Mecatrônica.

3.1 Métodos Aplicados

Este trabalho estudará os exemplos de referência do TensorFlow Lite e aplicará o método indutivo, que observa e analisa algum caso específico, para a partir disso concluir se esta ferramenta é aplicável no contexto de projetos mecatrônicos.

4 IMPLEMENTAÇÃO DE APRENDIZADO DE MÁQUINAS EM MICROCONTROLADORES

Ao longo deste projeto experimentou-se as aplicações de referência do TensorFlow Lite aplicados a múltiplas plataformas de desenvolvimento microcontroladas. Foram desenvolvidos dois exemplos utilizando-se a ferramenta de desenvolvimento da Google, chamado de Google Colaboratory (GOOGLE, 2022a), que são traduções dos exemplos de referencias originais *hello_world* e *micro_speech*, expandidos para que possuam uma explicação mais abrangente dos elementos que compõe uma aplicação embarcada usando o TensorFlow Lite.

As plataformas de desenvolvimento testadas são apresentadas na tabela 4.

Tabela 2 – Plataformas de desenvolvimento adquiridas para teste

Plataforma	CPU Flash	SRAM	Núcleo	Preço
Arduino Nano 33 BLE Sense	1MB	256KB	Cortex M4 32-bit, 64MHz	U$ 43,50
SparkFun Edge	1MB	348KB	Cortex-M4F 32-bit, 48MHz	U$ 16,50
STM32F746 Discovery Kit	1MB	340KB	Cortex-M7 32-bit, 216 MHz	U$ 55,12
Raspberry Pi Pico Seeed Studio	2MB	264KB	Cortex M0+ 32-bit 133MHz	U$ 4,00

Fonte: Elaboração própria

4.1 Casos de uso estudados

Foram desenvolvidos dois casos de uso, um exemplo básico onde descrevemos, de forma completa, desde os princípios e a metodologia para desenvolver uma rede neural capaz de interpretar a função seno até a geração de código de máquina embarcável. No outro caso exemplo utiliza-se o microfone para desenvolver uma aplicação de reconhecimento de voz.

4.2 Caso de uso: aprendizado de máquina para função seno

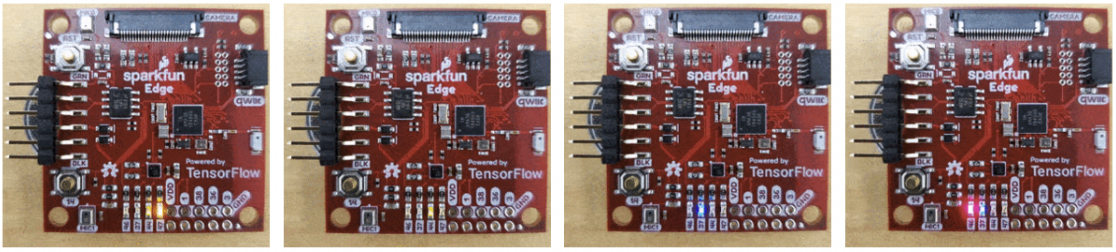
Este exemplo segue o que foi descrito no exemplo de referência *hello_world* do TensorFlow Lite e também utilizou as informações passadas pelo livro (WARDEN, 2019). Este exemplo de referência apresenta como ensinar redes neurais para interpretar uma função, neste caso a função seno, com um passo a passo para transformar a mesma em código embarcável para microcontroladores que suportam o TensorFlow

Lite. Apesar de parecer genérico, este exemplo pode ser aplicado no contexto de qualquer sinal, servindo como base para aplicações nas quais se conhece o espectro do sinal do processo que deseja-se estudar e pode ser expandido para detectar qualquer forma de onda ou sinal que se deseje controlar, medir ou avaliar.

Este modelo foi embarcado em duas plataformas microcontroladas distintas: ESP32 e Spark Fun. Na primeira, apresenta-se uma senoide sendo construída utilizando o modelo gerado pelo TensorFlow Lite na tela LCD da plataforma. Na plataforma Spark Fun apresenta-se a mesma senoide utilizando os LED's da plataforma.

A figura 10 apresenta o resultado deste modelo embarcado na plataforma Spark Fun, que mostra pelos LEDs ligados quando detecta os picos e vales da função seno.

Figura 2 – LEDs da SparkFun apresentando o modelo gerado sendo executado



Fonte: Do próprio autor. Cada LED representa uma faixa do seno: da esquerda para a direita entre -1 e -0.5, entre - 0.5 e 0, entre 0 e 0.5 e finalmente entre 0.5 e 1 respectivamente.

O Anexo 1 apresenta o software desenvolvido no Google Colaboratory, traduzido e ampliado do original, que descreve o processo, desde a geração do modelo até a geração do código de máquina a ser embarcado.

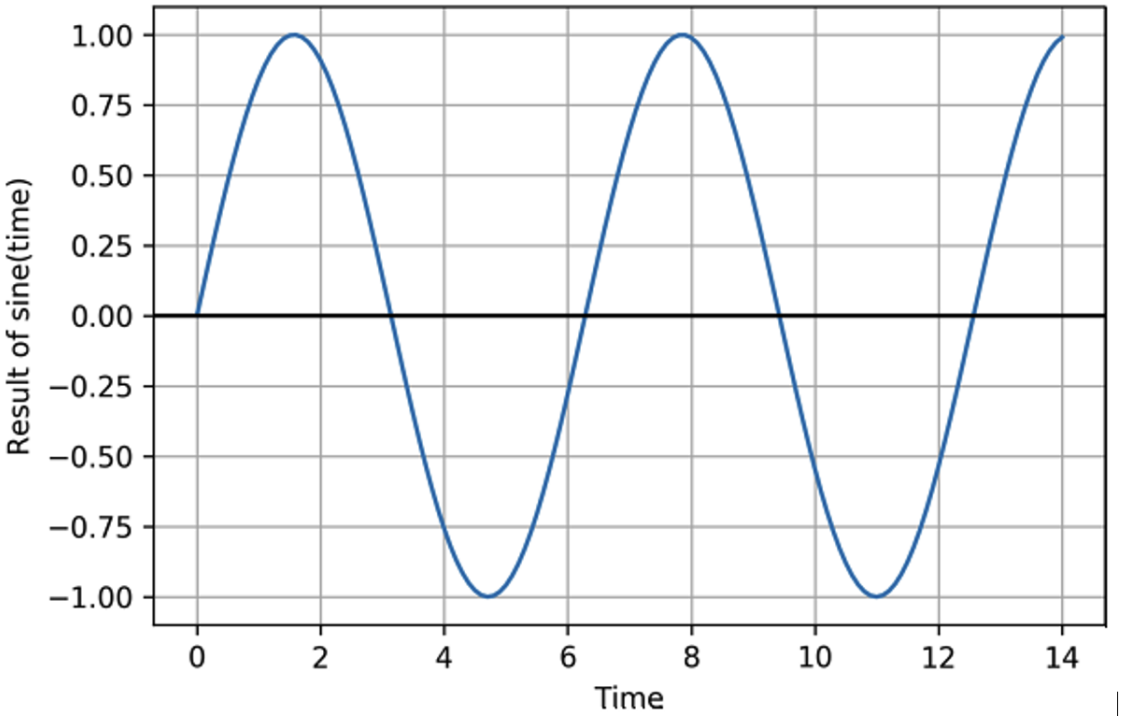
4.2.1 Construindo e treinando o modelo deste caso de uso

Colocando o conhecimento adquirido do estudo realizado no capítulo 2 em prática, inicia-se a construção e treinamento de um modelo, em seguida, o mesmo é integrado a um um programa de microcontrolador.

Para criar nosso modelo de aprendizado de máquina, em linguagem de programação Python, precisamos das bibliotecas TensorFlow e do Google Colaboratory, que é utilizado como um bloco de anotações interativo, baseado em nuvem computacional, para experimentar código Python. Essas são algumas das ferramentas mais importantes para engenheiros de aprendizado de máquina do mundo real e todas elas são gratuitas.

Na fundamentação teórica nós discutimos como as redes de aprendizado profundo aprendem a modelar padrões em seus dados de treinamento para que possam fazer previsões. Os dados que servirão para treinamento da rede são provenientes de uma onda senoidal, conforme a figura 11.

Figura 3 – Função seno



Fonte: WARDEN, 2019, pg. 26

O objetivo é treinar um modelo que possa tomar um valor x, e prever seu seno(x), ou seja, o valor y. Em um aplicativo do mundo real, se você precisasse do seno de x, você poderia simplesmente calculá-lo diretamente. No entanto, ao treinar um modelo para aproximar o resultado, podemos demonstrar os conceitos básicos de aprendizado de máquina.

4.2.2 Utilizando Python e Jupiter Notebooks no desenvolvimento

Python é a linguagem de programação favorita de cientistas e engenheiros de aprendizado de máquina. É fácil de aprender, funciona bem para muitas aplicações diferentes e tem muitas bibliotecas para tarefas úteis que envolvem dados e operações matemáticas. A maior parte da pesquisa de aprendizado profundo é feita usando Python, e os pesquisadores geralmente liberam o código-fonte Python para os modelos que criam.

Python é especialmente bom quando combinado com algo chamado Jupyter Notebooks. Este é um formato de documento especial que permite misturar escrita, gráficos e código que podem ser executados com o clique de um botão. Os blocos de anotações do Jupyter Notebooks são amplamente usados para descrever, explicar e explorar códigos e problemas de aprendizado de máquina.

Para executar nosso notebook, usaremos uma ferramenta chamada Colaboratory, ou Colab para abreviar. Ela é fornecida gratuitamente como uma ferramenta

para incentivar a pesquisa e o desenvolvimento em aprendizado de máquina. O Colab permite que se execute *Notebooks* no hardware da Google, a custo zero. Você pode até mesmo configurar o Colab para executar seu código em hardware dedicado que pode executar o treinamento mais rápido do que em um computador normal.

4.2.3 Usando a API Keras do TensorFlow

Nesta seção, utiliza-se o Keras, que é uma API de alto nível do TensorFlow e que facilita a criação e o treinamento de redes de aprendizado profundo. Também usaremos o TensorFlow Lite, que é um conjunto de ferramentas para implantar modelos do TensorFlow em dispositivos móveis e embarcados, para executar nosso modelo no microcontrolador.

4.2.4 Construindo o modelo

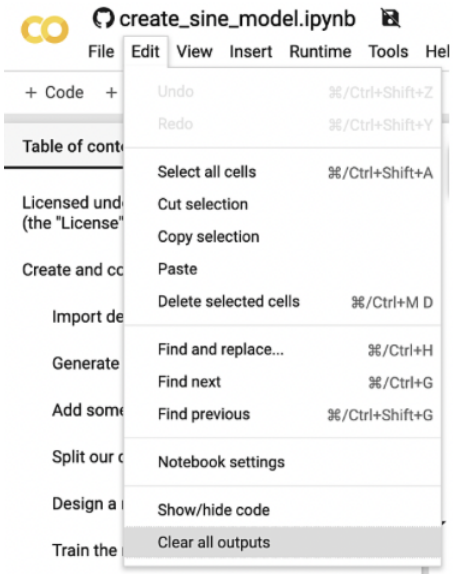
No processo de construção, treinamento e conversão de nosso modelo, devemos, primeiro, abrir o Colab com o Notebook. Depois que a página for carregada, na parte superior, deve-se clicar em "Executar no Google Colab"conforme a figura 12.

Figura 4 – Executar o Jupiter Notebook no Google Colab

Fonte: WARDEN, 2019, pg. 22

Se o Notebook já havia sido executado, deve-se limpar os dados para que o Notebook inicialize zerado. Para fazer isso, no menu do Colab, clique em Editar e, em seguida, selecione "Clear all outputs"conforme mostra a figura 13.

Figura 5 – Reiniciando o Jupiter Notebook resetando dados anteriores



Fonte: WARDEN, 2019, pg. 23

4.2.5 Importando dependências

Deve-se instalar a biblioteca TensorFlow 2.0 assim como diversas bibliotecas matemáticas do TensorFlow, NumPy, Matplotlib e o próprio Python.

Uma vez executado o comando de instalação das bibliotecas Python (comumente conhecidas como comandos pip), as dependências começarão a ser instaladas e você verá algumas saídas aparecendo. Ao final do processo, uma mensagem informando se a instalação foi realizada com sucesso será informada e a partir daí, é possível utilizar O Keras.

4.2.6 Gerando dados

As redes de aprendizagem profunda aprendem a modelar padrões mediante o seu treinamento com uma série de dados. Vamos treinar uma rede para modelar dados gerados por uma função senoidal. Isso resultará em um modelo que pode tomar um valor, x, e prever seu seno, y.

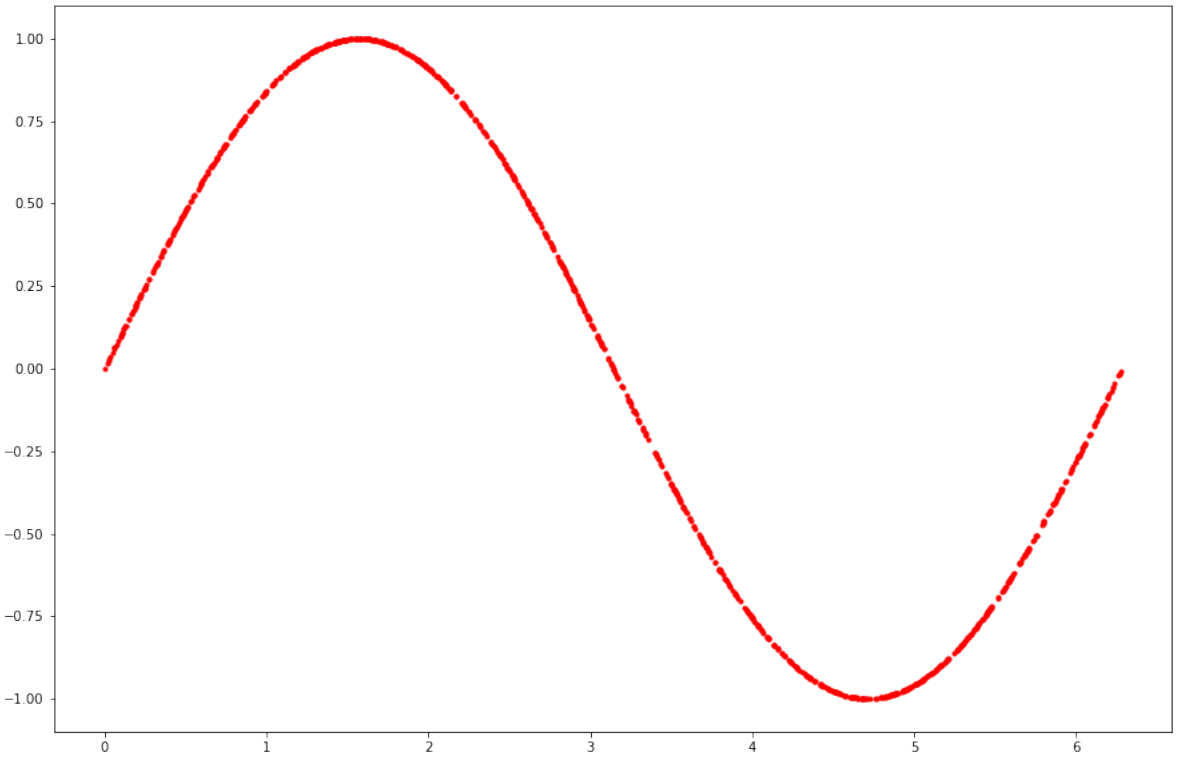
Antes de prosseguir, precisa-se de alguns dados que representem o valor senoidal. Em uma situação do mundo real, podemos estar coletando dados de sensores ou de registradores. Para este exemplo, no entanto, será gerado um conjunto de dados por meio de um software.

O plano é gerar 1.000 valores que representam pontos aleatórios ao longo de uma onda senoidal. Cada ciclo completo de uma onda é chamado de período. A partir do gráfico, podemos ver que um ciclo completo é completado aproximadamente a cada seis unidades no eixo x. De fato, o período de uma onda senoidal é de 2. Para que tenhamos uma onda senoidal completa de dados para treinar, nosso código gerará

valores x aleatórios de 0 a 2. Em seguida, ele calculará o seno para cada um desses valores. O código em Python usando a biblioteca NumPy que gera esses dados é mostrado na figura 14.

```
# O conjunto de dados ira possui a quantidade de SAMPLES
SAMPLES = 1000
# Define um valor SEED para que possamos ter o mesmo resultado de número
# aleatórios se o código for executado novamente. O valor de semente é
# um valor base usado por um gerador pseudo-aleatório para produzir
# números aleatórios.
SEED = 1200
np.random.seed(SEED)
tf.random.set_seed(SEED)
# Gere um conjunto uniformemente distribuído de números aleatórios
# no intervalo de 0 to 2, com quantidade de pontos definidas como SAMPLES.
x_values = np.random.uniform(low=0, high=2*math.pi, size=SAMPLES)
# Embaralhe os valores para garantir que não estão em ordem.
np.random.shuffle(x_values)
# Calcule os valores de seno correspondentes
y_values = np.sin(x_values)
# Plote os resultados
plt.rcParams['figure.figsize'] = [15, 10]
plt.plot(x_values, y_values, 'r.')
plt.show()
```

Figura 6 – Plotagem de dados gerados represetando a função seno

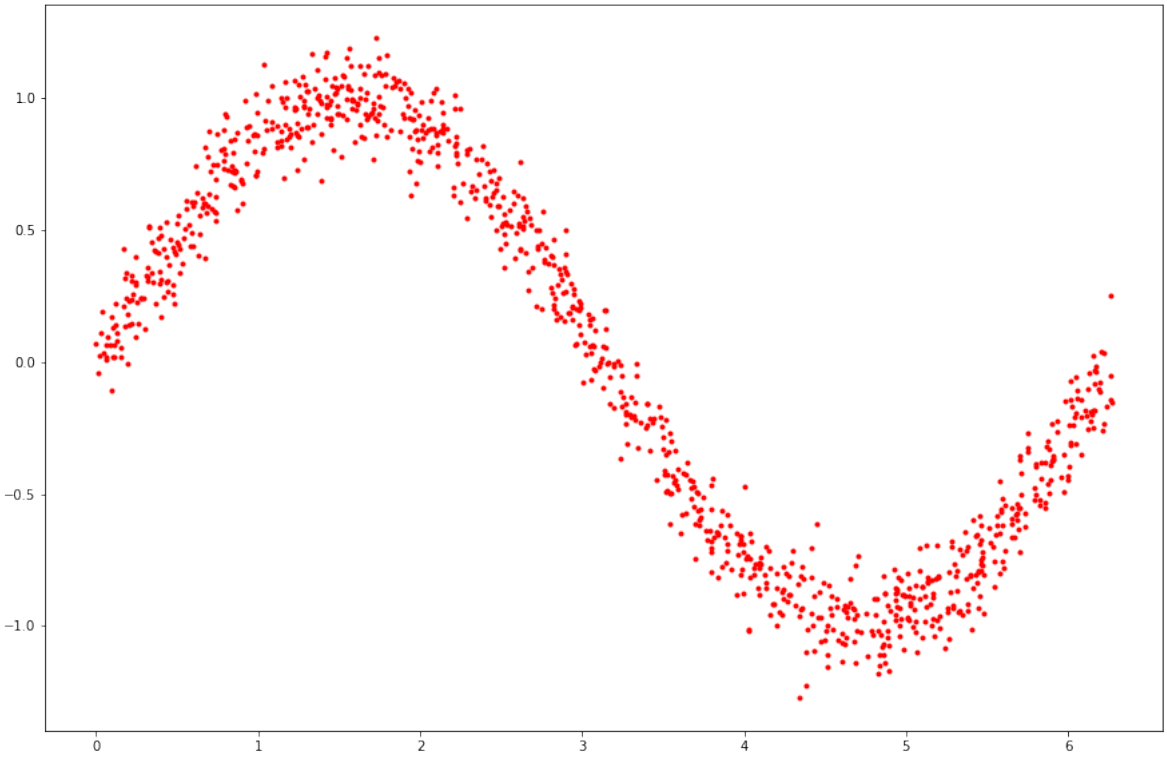


Fonte: Do autor (2022).

Uma das coisas interessantes sobre as redes de aprendizagem profunda é a sua capacidade de peneirar padrões de ruído. Isso permite que eles façam previsões, mesmo quando treinados em dados confusos do mundo real. Para demostrar, adiciona-se algum ruído aleatório aos dados, mostrado na figura 15.

```
# Adicione um pequeno número aleatório a cada valor de y
y_values += 0.1 * np.random.randn(*y_values.shape)
# Plotar nossos dados
plt.rcParams['figure.figsize'] = [15, 10]
plt.plot(x_values, y_values, 'r.')
plt.show()
```

Figura 7 – Função seno com ruído adicionado



Fonte: Do autor (2022).

4.2.7 Dividindo os dados

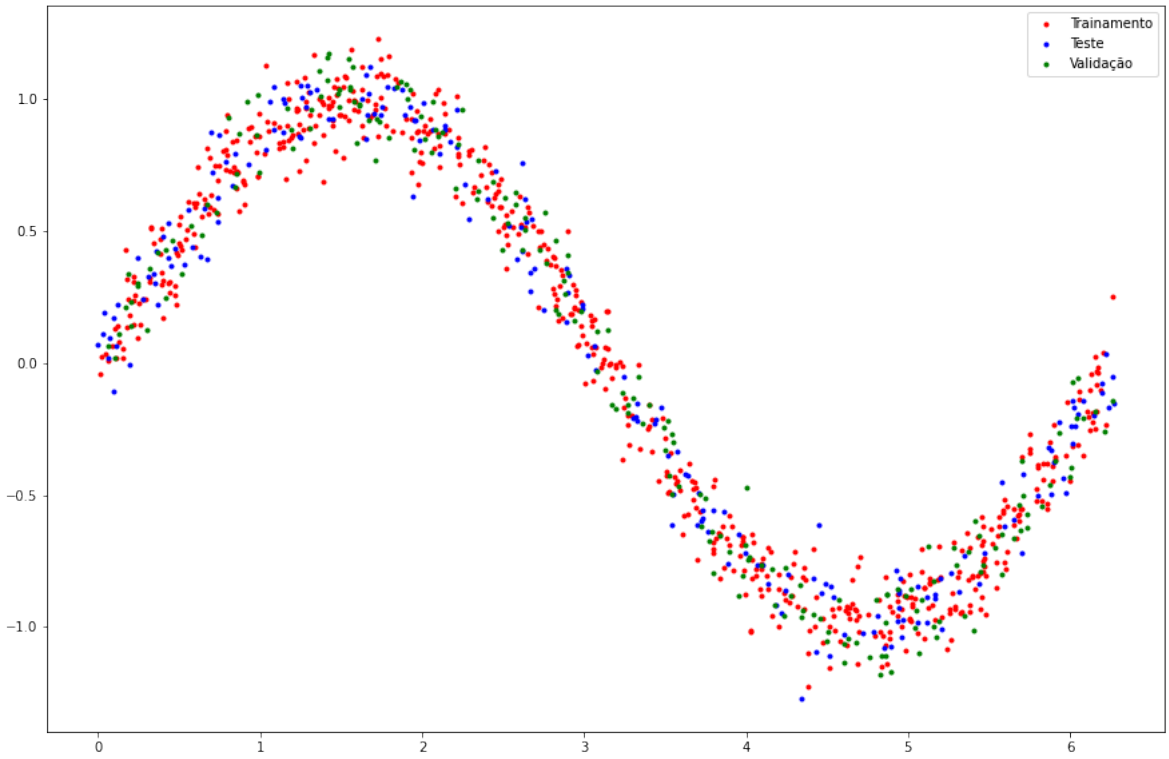
Para avaliar a precisão do modelo que treinamos, precisamos comparar suas previsões com dados reais e verificar o quão bem elas correspondem. Essa avaliação acontece durante o treinamento (onde é referido como validação) e após o treinamento (referido como teste). É importante, em cada caso, que usemos dados novos que ainda não foram usados para treinar o modelo.

Para garantir que tenhamos dados para usar para avaliação, deixaremos alguns dados de lado antes de começarmos o treinamento. Reserva-se 20% dos dados para validação e outros 20% para testes. Usar-se-á os 60% restantes para treinar o

modelo. Essa é uma divisão típica usada ao treinar modelos. O código abaixo apresenta como dividir os dados e a Figura 16 mostra o resultado obtido.

```
# Usaremos 60% de nossos dados para treinamento e 20% para teste.
# Os 20% restantes será usado para validação.
TRAIN_SPLIT =  int(0.6 * SAMPLES)
TEST_SPLIT = int(0.2 * SAMPLES + TRAIN_SPLIT)
# Use np.split para dividir nossos dados em três partes.
# O segundo argumento para np.split é um array de índices onde os dados
# serão divididos. Como existem dois índeces, os dados serão divididos
# em três pedaços.
x_train, x_validate, x_test = np.split(x_values, [TRAIN_SPLIT, TEST_SPLIT])
y_train, y_validate, y_test = np.split(y_values, [TRAIN_SPLIT, TEST_SPLIT])
# Verifique se nossas divisões somam corretamente
assert (x_train.size + x_validate.size + x_test.size) ==  SAMPLES
# Plotar os dados
plt.rcParams['figure.figsize'] = [15, 10]
plt.plot(x_train, y_train, 'r.', label="Treinamento")
plt.plot(x_test, y_test, 'b.', label="Teste")
plt.plot(x_validate, y_validate, 'g.', label="Validação")
plt.legend()
plt.show()
```

Figura 8 – Divisão dos dados para treinamento, validação e teste



Fonte: Do autor (2022).

Observe que as cores separam os dados em dados para treinamento (em vermelho), dados para validação (em azul) e para teste (em verde). Observe que os dados devem sempre representar todo o fenômeno em estudo.

4.2.8 Definindo um modelo básico

O objetivo é ter um modelo capaz de prever, a partir de um valor de entrada (neste caso, x), o valor de saída numérico (o seno de x). Esse tipo de problema é chamado de Regressão, e em geral, pode-se usar modelos de regressão para todos os tipos de dados que exigem uma saída numérica. Por exemplo, um modelo de Regressão pode tentar prever a velocidade de corrida de uma pessoa com base em dados de um acelerômetro.

Usando-se Keras, que é uma API de alto nível do TensorFlow para a criação de redes de aprendizado profundo, essa tarefa é facilitada:

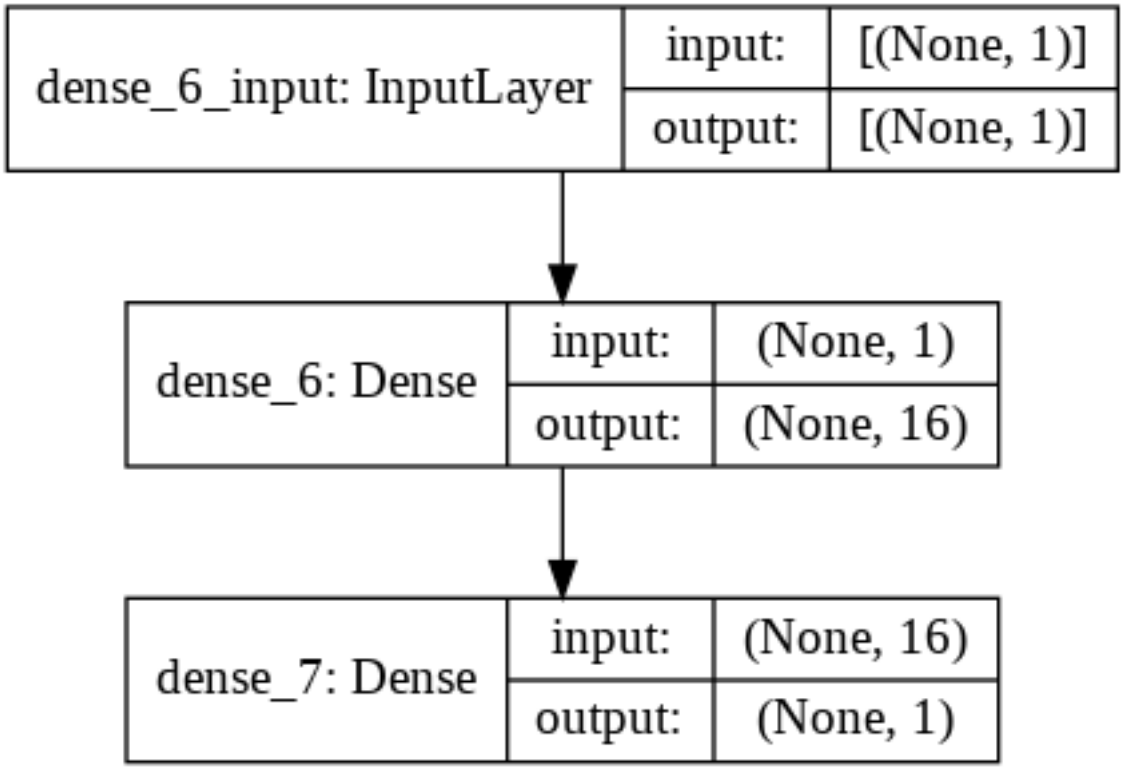
```
# Usaremos o Keras para criar uma arquitetura de modelo simples
from tf.keras import layers
model_1 = tf.keras.Sequential()
# A primeira camada recebe uma entrada escalar e a alimenta através de
# 16 "neurônios". Os neurônios decidem se ativam com base na função de
# ativação 'relu'.
model_1.add(layers.Dense(16, activation='relu', input_shape=(1,)))
# A camada final é um único neurônio, já que queremos produzir um único valor.
model_1.add(layers.Dense(1))
# Compile o modelo usando um otimizador padrão e uma função de perda
# para regressão.
model_1.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])
# Imprima a arquitetura do modelo.
tf.keras.utils.plot_model(model_1, show_shapes=True, show_layer_names=True)
```

Primeiro, cria-se um modelo sequencial usando Keras, o que significa apenas um modelo em que cada camada de neurônios é empilhada em cima da próxima. Em seguida, define-se duas camadas.

A primeira camada tem uma única entrada – nosso valor x – e 16 neurônios. É uma camada densa (também conhecida como camada totalmente conectada), o que significa que a entrada será alimentada em cada um de seus neurônios durante a inferência, quando estamos fazendo previsões. Cada neurônio será então ativado até certo ponto.

A quantidade de ativação para cada neurônio é baseada em seus valores de peso e viés, aprendidos durante o treinamento, e sua função de ativação. A ativação do neurônio é a saída numérica. A figura 17 apresenta o plotagem realizada pelo Keras do modelo desenvolvido.

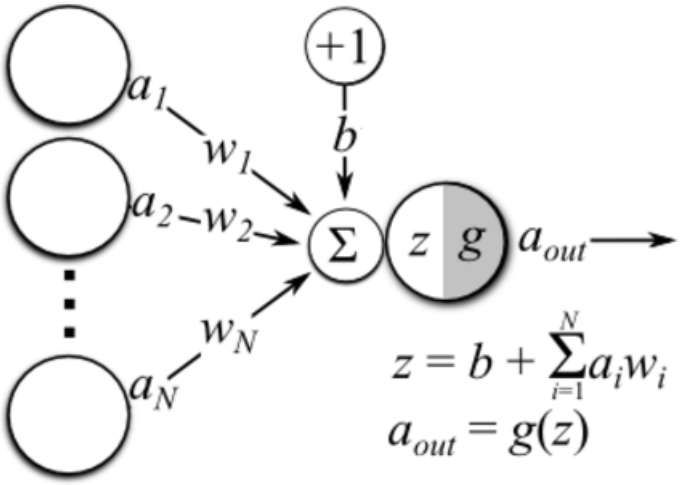
Figura 9 – Arquitetura do modelo



Fonte: Do autor (2022).

A ativação é calculada por uma fórmula simples. Nunca precisaremos codificar isso nós mesmos, uma vez que é tratado pelo Keras e pelo TensorFlow, mas será útil saber à medida que avançamos no aprendizado profundo.

Figura 10 – Pesos e vieses na ativação de um neurônio



Fonte: Adaptado de (AI WIKI, 2022).

Para calcular a ativação do neurônio, sua entrada é multiplicada pelo peso, e por fim o viés (*bias*) é adicionado ao resultado. O valor calculado é passado para

a função de ativação. O número resultante é usado como entrada pela função de ativação do neurônio. A figura 18 apresenta de forma simplificada os pesos *w* e o viés *b*, apresentando a forma como ele é calculado para posteriormente ser passado para a função de ativação.

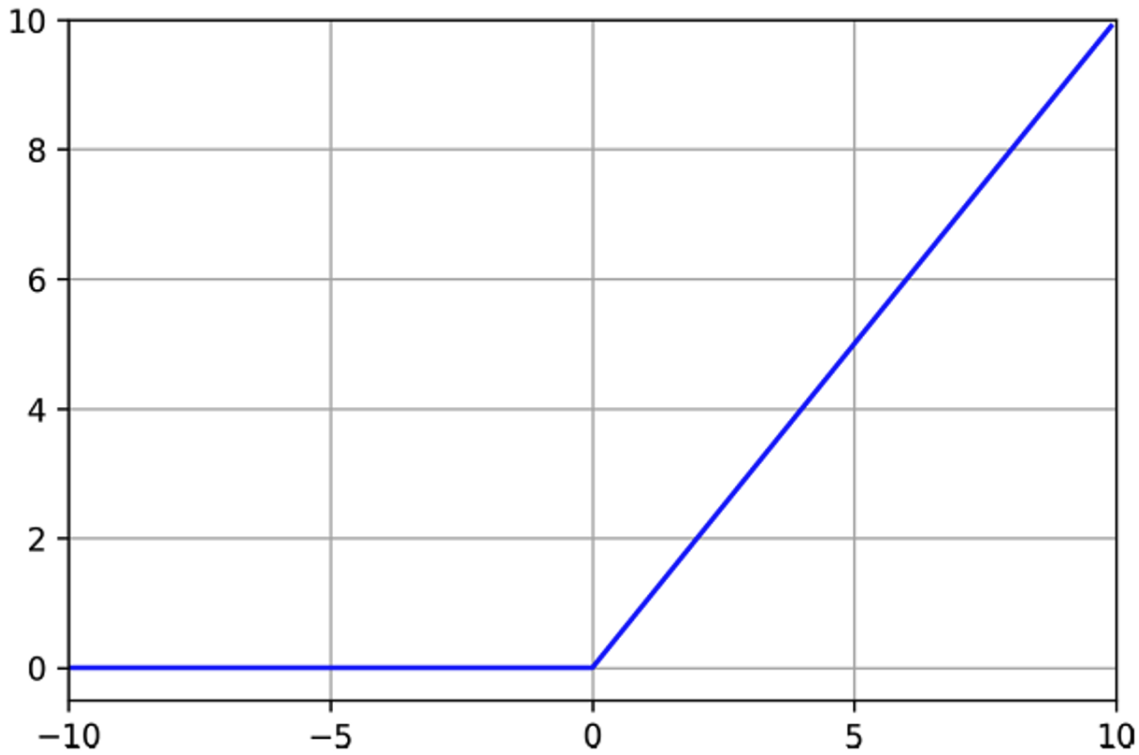
A função de ativação é uma função matemática usada para moldar a saída do neurônio. Em nossa rede, estamos usando uma função de ativação chamada unidade linear retificada, ou ReLU para abreviar. Isso é especificado em Keras pelo argumento "activation=relu".

ReLU é uma função simples, que pode ser definida em Python como:

```
def relu(input): return max(0.0, input)
```

ReLU retorna o que for o valor maior: sua entrada ou zero. Se seu valor de entrada for negativo, o ReLU retornará zero. Se seu valor de entrada estiver acima de zero, o ReLU o retornará inalterado. A figura 19 apresenta a relação entre a entrada e a saída da função de ativação ReLU.

Figura 11 – Relação de entrada vs. saída da função ReLU



Fonte: WARDEN, 2019, pg. 29

Sem uma função de ativação, a saída do neurônio seria sempre uma função linear de sua entrada. Isso significaria que a rede poderia modelar apenas relações lineares nas quais a razão entre x e y permanece a mesma em toda a faixa de valores.

Isso impediria uma rede de modelar nossa onda senoidal porque uma onda senoidal é não-linear.

Como o ReLU é não-linear, ele permite que várias camadas de neurônios unam forças e modelem relações não lineares complexas, nas quais o valor y não aumenta na mesma quantidade para cada incremento de x.

Existem outras funções de ativação, mas o ReLU é o mais comumente usado. Cada função de ativação tem diferentes compensações, e os engenheiros de aprendizado de máquina experimentam para descobrir quais opções funcionam melhor para uma determinada arquitetura.

Os números de ativação da nossa primeira camada serão alimentados como entradas para a nossa segunda camada. Como essa camada é um único neurônio, ela receberá 16 entradas, uma para cada um dos neurônios da camada anterior. Sua finalidade é combinar todas as ativações da camada anterior em um único valor de saída. Como essa é a nossa camada de saída, não especificamos uma função de ativação – queremos apenas o resultado bruto.

Como esse neurônio tem várias entradas, ele tem um valor de peso correspondente para cada uma. A saída do neurônio é calculada pela seguinte fórmula, mostrada em Python:

```
# Aqui ‘inputs‘ e ‘weights‘ são arrays NumPy com 16 elementos cada
output = sum((inputs * weights)) + bias
```

O valor de saída é obtido multiplicando cada entrada com seu peso correspondente, somando os resultados e, em seguida, adicionando o viés do neurônio.

4.2.9 Treinando o modelo

Agora devemos realizar o treinamento e, em seguida, avaliar o desempenho para ver o quão bem funciona, usando métricas que permitem decidir se é bom o suficiente, ou se devemos fazer alterações em nosso modelo e treiná-lo novamente. Para treinar um modelo em Keras, chamamos apenas o método *fit()*, passando todos os nossos dados e alguns outros argumentos importantes.

Alguns argumentos importantes passados para o método de ajuste são o *x_train* e *y_train*, que é essencialmente os valores dos dados de treinamento. Em seguida, as épocas especificam quantas vezes todo o nosso conjunto de treinamento será executado através da rede durante o treinamento. Quanto mais épocas, mais treinamento ocorrerá.

Outro argumento importante é o *batch_size* que especifica quantos dados de treinamento devem ser alimentados na rede antes de medir sua precisão e atualizar seus pesos e vieses. Se quiser, pode-se especificar uma *batch_size* de 1, o que significa

que executaríamos a inferência em um único ponto de dados, mediríamos a perda da previsão da rede, atualizaríamos os pesos e vieses para tornar a previsão mais precisa da próxima vez e, em seguida, continuaríamos esse ciclo para o resto dos dados.

Se tivermos 600 pontos de dados, cada época resultará em 600 atualizações na rede, exigindo muito tempo computacional. Se definir *batch_size* para 600, cada lote incluiria todos os nossos dados de treinamento. Agora teríamos que fazer apenas uma atualização na rede a cada época, o que é muito mais rápido.

Finalmente, ele usa o *validation_data* para especificar o conjunto de dados de validação que será executado através da rede durante todo o processo de treinamento, e as previsões da rede serão comparadas com os valores esperados.

4.2.10 Métricas do treinamento

Depois de executar o treinamento, pode-se verificar algumas métricas para entender o resultado. As principais métricas importantes são a perda *(loss)*, erro absoluto médio *(mae)*, perda na validação *(val_loss)* e erro absoluto médio na validação *(val_mae)*. Elas nos informam diferentes características.

A métrica de perda, ou *loss*, é a saída da nossa função de perda. Ele usa o erro quadrático médio, que é expresso como um número positivo. Geralmente, quanto menor o valor de perda, melhor, então isso é uma coisa boa coisa a se observar enquanto avaliamos o modelo.

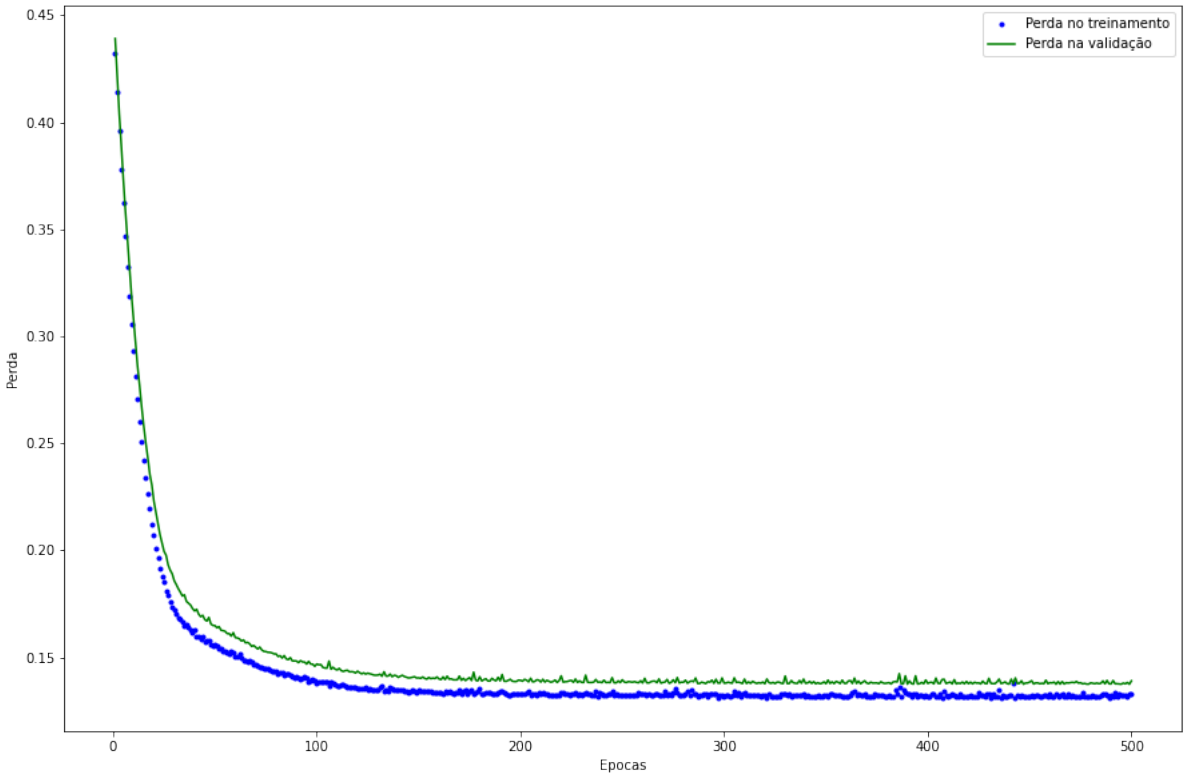
A métrica de erro absoluto médio, ou *mae*, é o erro absoluto médio de nossos dados de treinamento. Mostra a diferença média entre as previsões da rede e os valores y esperados dos dados de treinamento. Podemos esperar que nosso erro inicial seja ruim, uma vez que é baseado em uma rede não treinada.

O *val_loss* é a saída da função de perda em nossos dados de validação. Em nossa época final, a perda de treinamento (0,13) é ligeiramente menor do que a perda de validação (0,15). Esta é uma dica de que nossa rede pode estar se ajustando demais, porque está tendo um desempenho pior em dados que não viu antes.

E por último o *val_mae* é o erro absoluto médio para nossos dados de validação. Como o valor aproximado de 0,3 é pior do que o erro absoluto médio em nosso conjunto de treinamento, isso indica que a rede pode estar se ajustando demais.

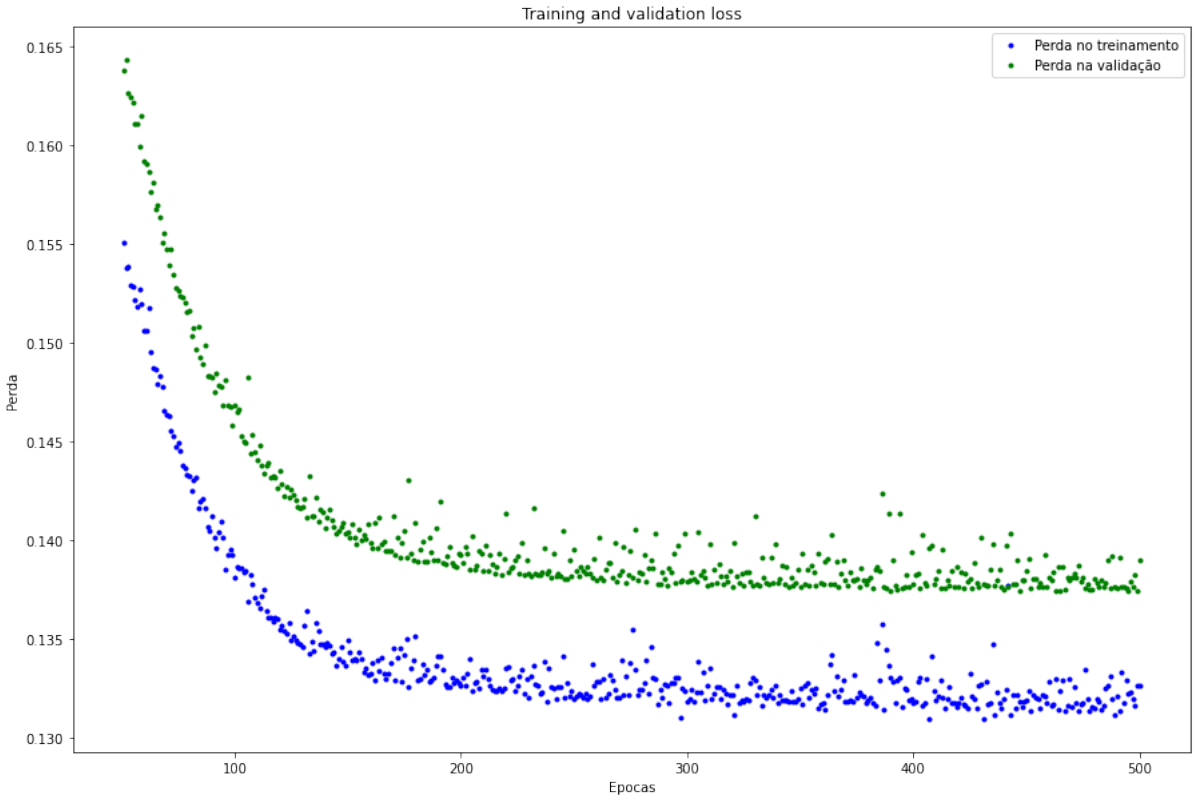
Na figura 20, a quantidade de perda diminui rapidamente ao longo das primeiras 50 épocas, antes de se achatar. Isso significa que o modelo está melhorando e produzindo previsões mais precisas. Geralmente, o objetivo é interromper o treinamento quando o modelo não estiver mais melhorando ou a perda de treinamento for menor do que a perda de validação.

Figura 12 – Gráfico comparando a perda do treinamento e validação



Fonte: Do autor (2022).

Figura 13 – Gráfico comparando a perda após 50 épocas



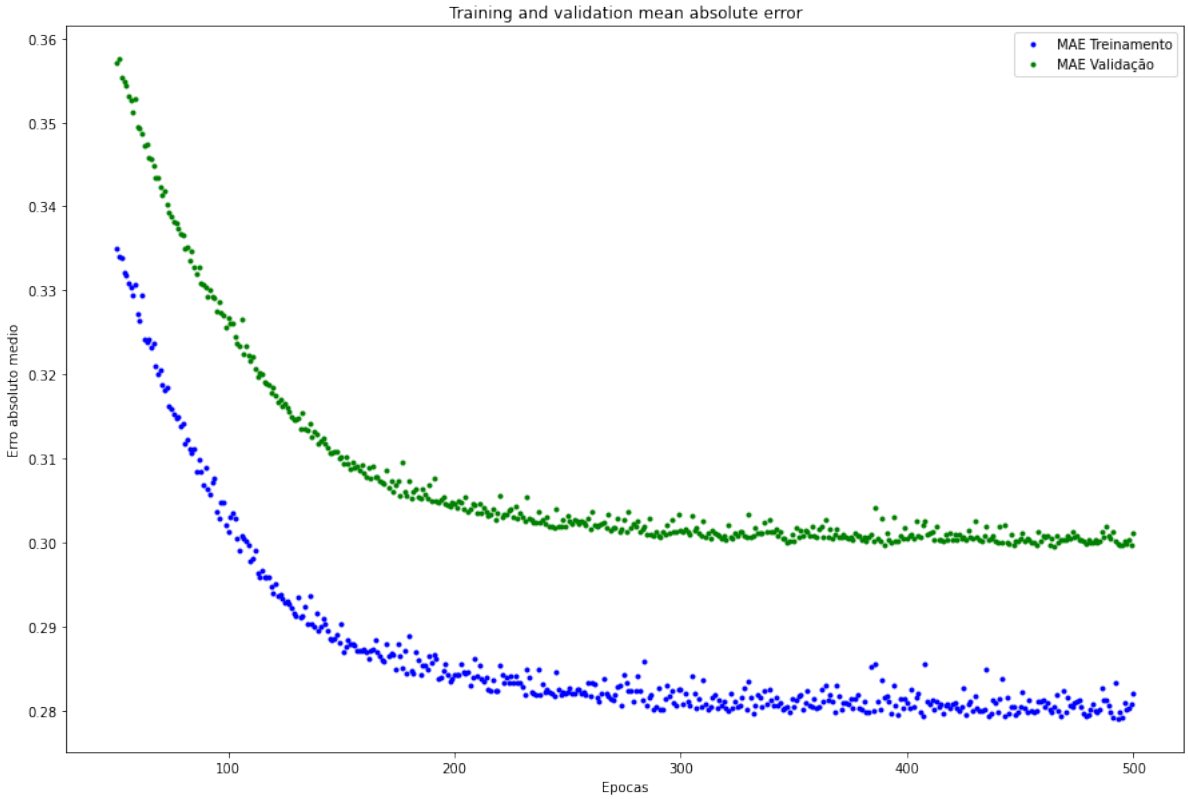
Fonte: Do autor (2022).

A perda cai vertiginosamente nas primeiras épocas, o que torna o resto do gráfico bastante difícil de ler. A figura 21 torna isto mais claro pois ignora as primeiras 50 épocas, faciliando a a visualização.

A perda continua a reduzir até cerca de 500 épocas, altura em que é maioritariamente estável. Isso significa que provavelmente não há necessidade de treinar nossa rede por tantas vezes. No entanto, pode-se perceber que o menor valor de perda ainda está em torno de 0,13, o que indica um erro de 13% nas vezes em que ele vai identificar a senoide. Além disso, os valores de perda de validação são consistentemente ainda maiores.

A Figura 22 mostra um gráfico do erro absoluto médio que nos fornece algumas pistas adicionais. Pode-se ver que, em média, os dados de treinamento mostram um erro menor do que os dados de validação, o que significa que a rede pode ter se ajustado demais ou aprendido os dados de treinamento de forma tão rígida que não pode fazer previsões eficazes sobre novos dados.

Figura 14 – Gráfico do erro absoluto do treinamento e validação



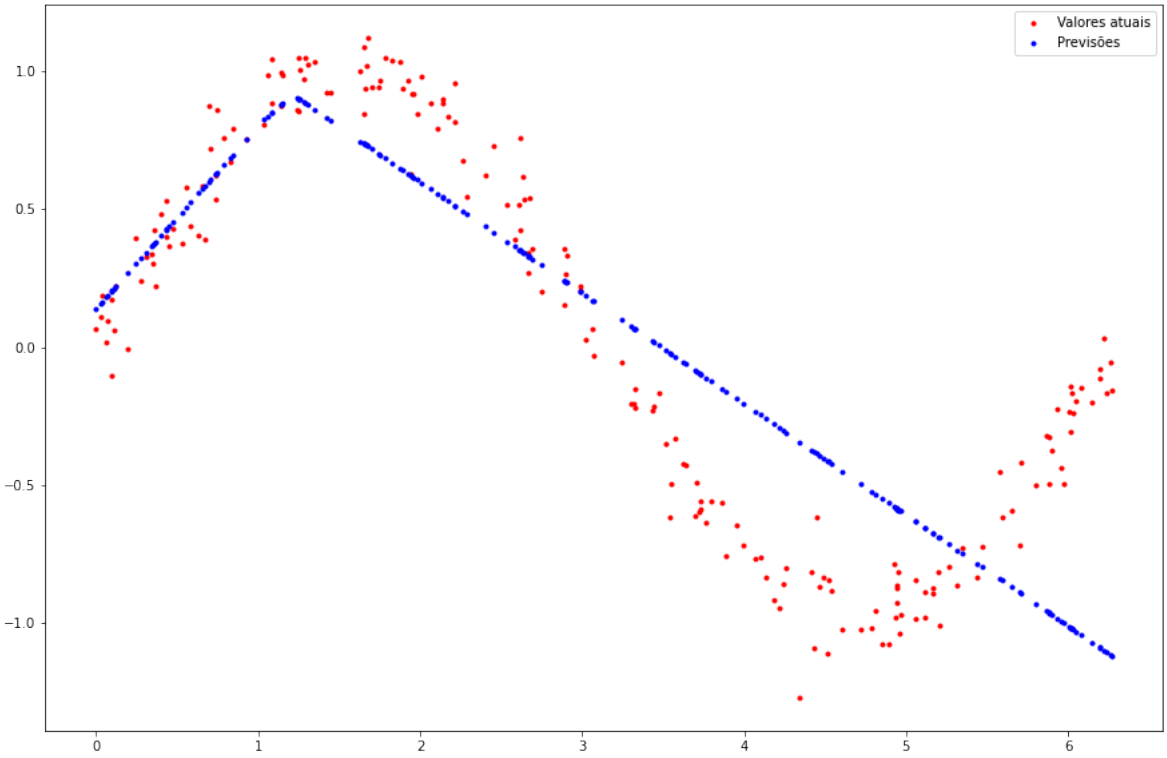
Fonte: Do auto (2022).

Além disso, os valores médios de erro absoluto são bastante altos, em torno de 0,3, o que significa que algumas das previsões do modelo estão erradas em pelo menos 30%. Como nossos valores esperados variam apenas em tamanho de -1 a +1, um erro de 0,3 significa que estamos muito longe de modelar com precisão a onda

senoidal.

Pode-se executar inferência em todos os valores x a partir dos dados de treinamento. O método retorna uma matriz de previsões. A figura 23 mostra um gráfico que deixa claro que a rede aprendeu a aproximar a função seno de uma forma muito limitada.

Figura 15 – Inferência apresentando má aprendizado da rede



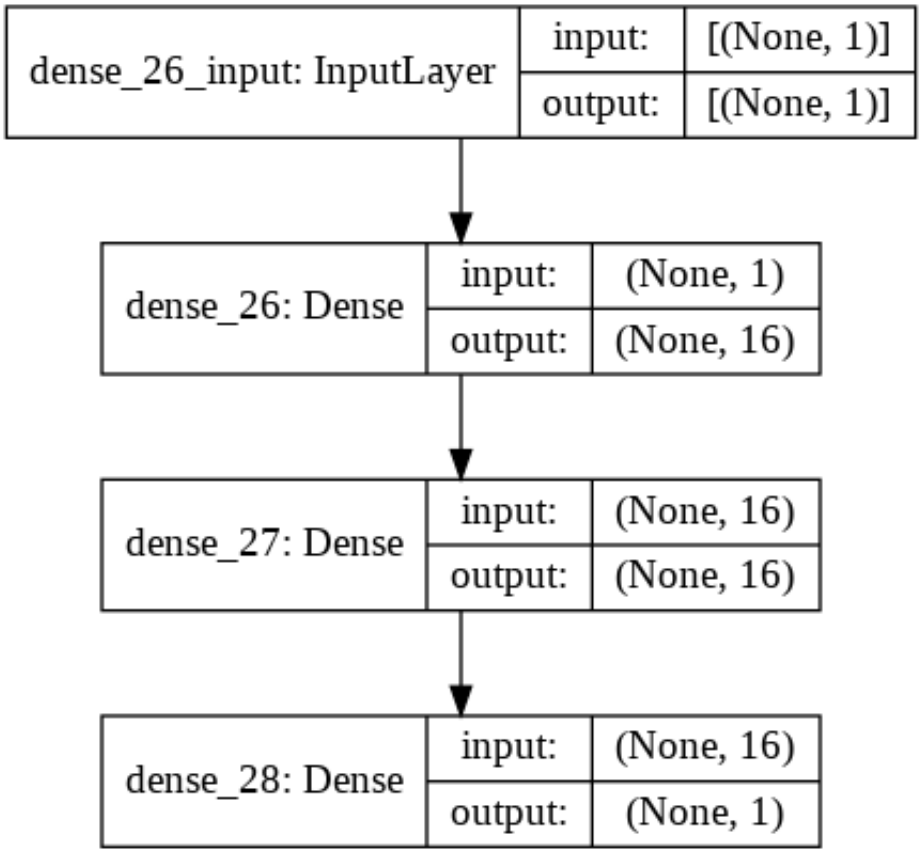
Fonte: Do autor (2022).

4.2.11 Melhorando o modelo

De acordo com a sugestão do autor, ele já mencionou anteriormente que a rede dada era muito pequena para aprender a complexidade dos dados da onda senoidal. Para impulsioná-lo, ele sugere o uso de mais camadas e neurônios.

Agora usaremos duas camadas de 16 neurônios e uma terceira camada com uma única saída, conforme realizado anteriormente. A figura 24 apresenta a nossa nova arquitetura.

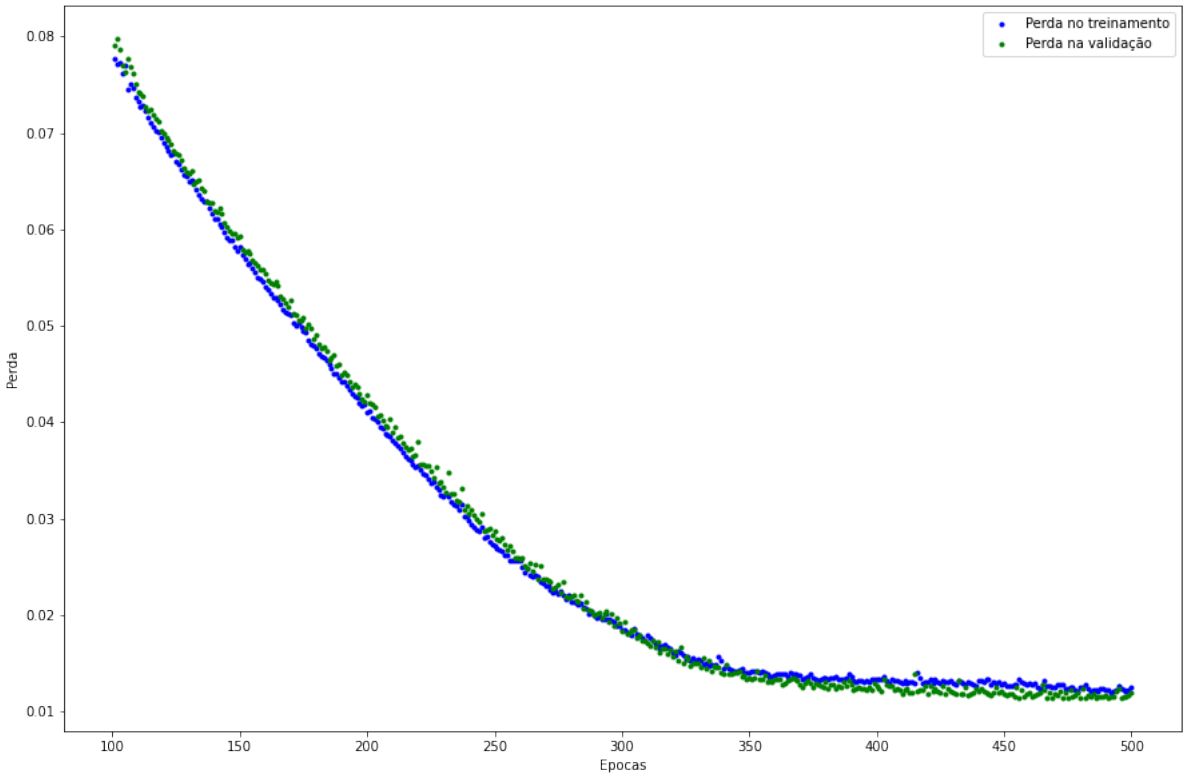
Figura 16 – Atualização da arquitetura do modelo



Fonte: Do autor (2022).

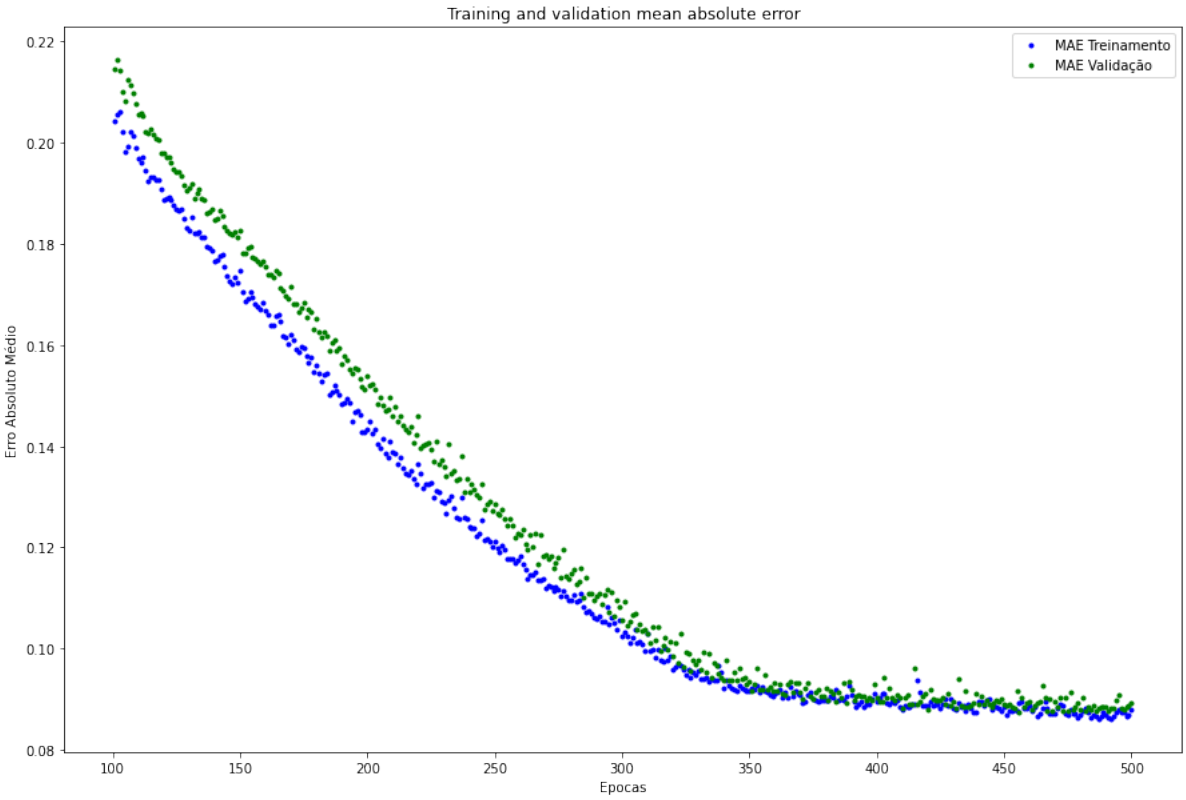
Depois de concluir o treinamento nesta nova arquitetura, agora é possível observar resultados melhores. A figura 25 mostra um gráfico de perda de treinamento e validação, pulando as primeiras 50 épocas. A figura 26 mostra a perda e o erro absoluto médio. Ambas as figuras apresentam dados removendo as primeiras 100 épocas.

Figura 17 – Perda após melhoria da rede neural



Fonte: Do autor (2022).

Figura 18 – Erro absoluto médio após melhoria da rede neural

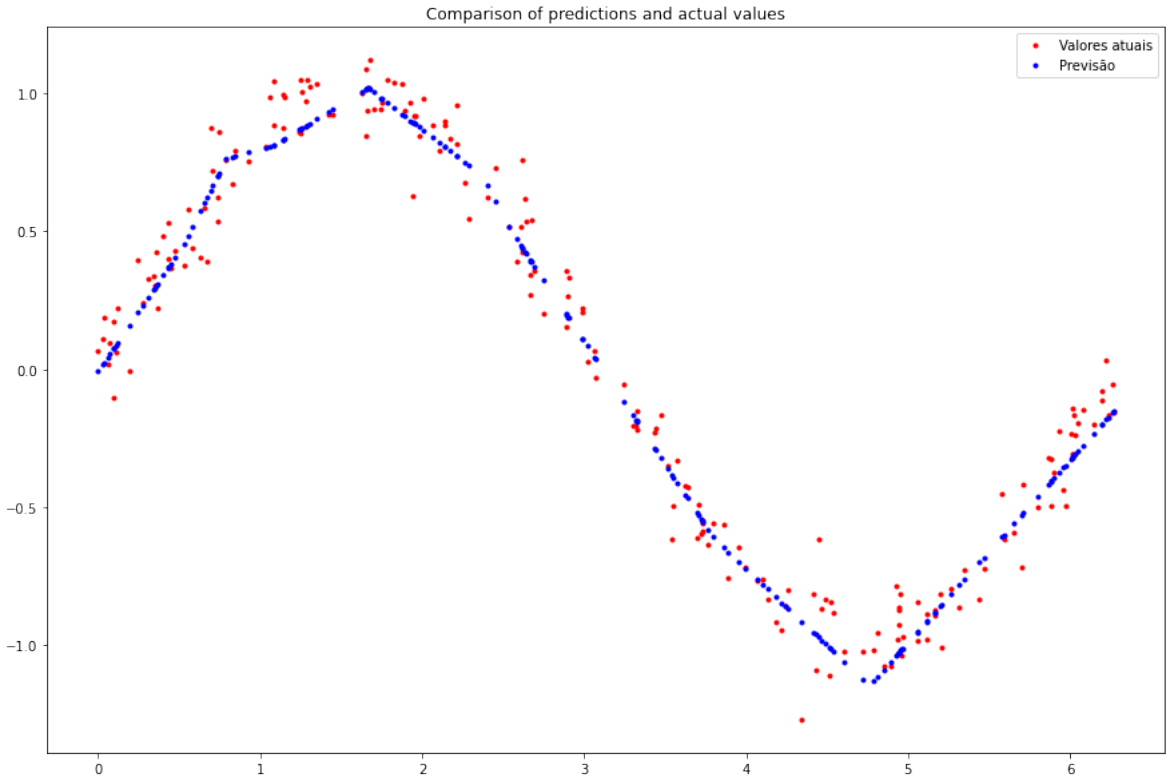


Fonte: Do autor (2022).

As métricas são melhores para validação do que para treinamento, o que significa que a rede não está se sobre-ajustando. A perda geral e o erro absoluto médio são muito melhores do que em nossa rede anterior.

Ao testar o modelo gerado usando nosso conjunto de dados de teste (20% dos pontos de dados não usados durante o treinamento) para prever nossa saída de modelo. A figura 27 mostra os resultados do modelo gerado.

Figura 19 – Resultado final após inferência do modelo atualizado



Fonte: Do autor (2022).

Percebe-se que, na maior parte, os pontos que representam os valores previstos e formam uma curva suave ao longo do centro da distribuição dos valores reais. Nossa rede aprendeu a se aproximar de uma curva senoidal, mesmo que o conjunto de dados estivesse com ruído.

4.2.12 Convertendo o modelo para TensorFlow Lite

Um modelo com precisão aceitável foi desenvolvido, agora, utilizremos o TensorFlow Lite Converter para converter o modelo em um formato compacto e com eficiência de memória.

Para implantar esse modelo em um microcontrolador, ele deve ser o menor possível. Para atingir isso, utilizamos a quantização, que é uma técnica para reduzir o tamanho de um modelo. Ela reduz a precisão dos pesos do modelo, e assim possivel-

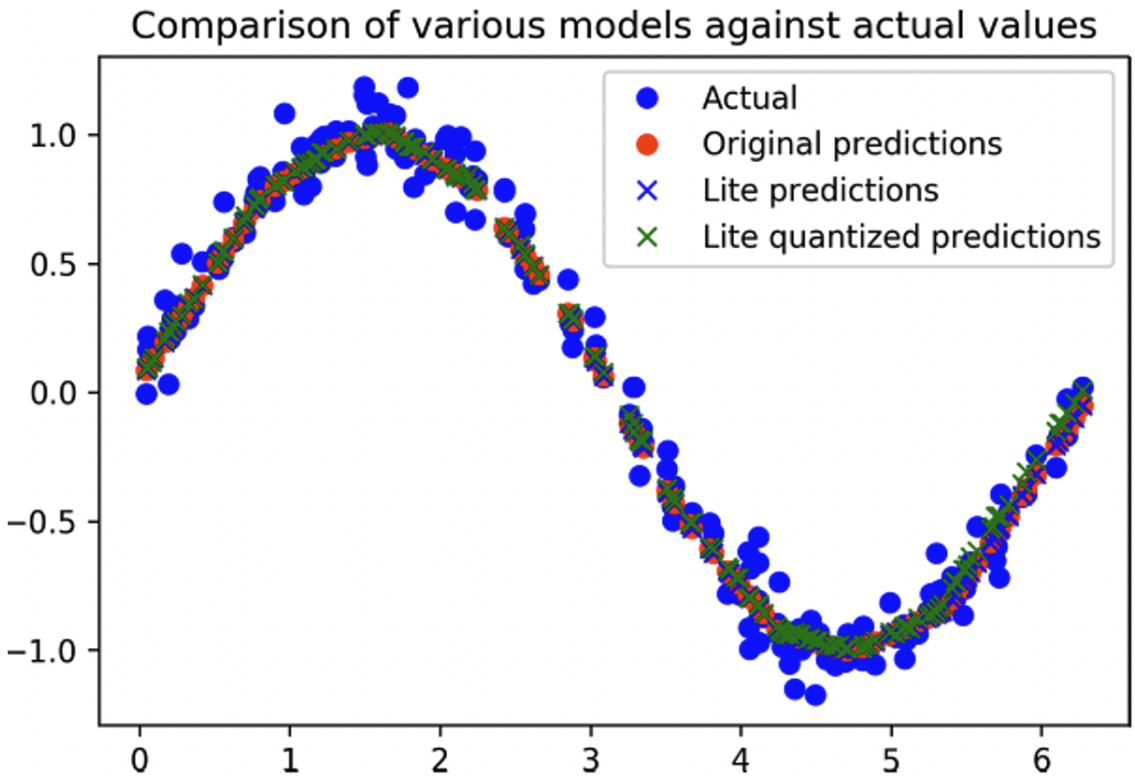
mente também as ativações (saídas de cada camada), economizando memória, com pouco impacto na precisão.

Como os cálculos necessários para modelos quantizados são mais simples, eles também são executados mais rapidamente. Na sequência, o modelo será convertido duas vezes: uma vez com quantização e outra sem.

```
# Converta o modelo para o formato TensorFlow Lite sem quantização
converter = tf.lite.TFLiteConverter.from_saved_model(MODEL_TF)
model_no_quant_tflite = converter.convert()

# Salva o modelo no disco
open(MODEL_NO_QUANT_TFLITE, "wb").write(model_no_quant_tflite)

# Converta o modelo para o formato TensorFlow Lite com quantização
def representative_dataset():
  for i in range(500):
    yield([x_train[i].reshape(1, 1)])
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.target_spec.supported_ops =
    [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
converter.inference_input_type = tf.int8
converter.inference_output_type = tf.int8
# Forneça um conjunto de dados representativo
# para garantir a quantização correta.
converter.representative_dataset = representative_dataset
model_tflite = converter.convert()

# Salve o modelo no disco
open(MODEL_TFLITE, "wb").write(model_tflite)
```

A Figura 28 mostra um gráfico comparando as previsões do modelo com os valores reais depois de gerar o código incorporável para o TensorFlow Lite. É um passo importante porque está reduzindo o tamanho do modelo, reduzindo assim sua precisão, o que pode adicionar erros.

Figura 20 – Resultado da inferência após quantização do modelo



Fonte: WARDEN, 2019, pg. 48

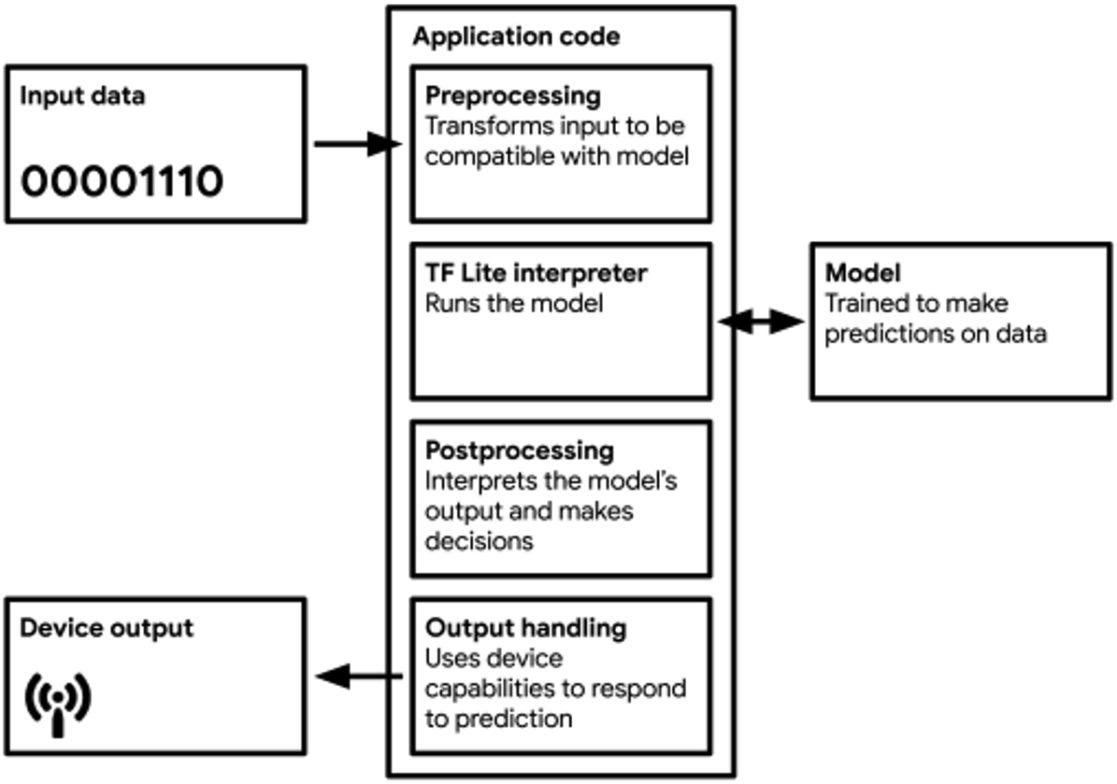
Percebe-se a partir do gráfico que as previsões para o modelo original, o modelo convertido e o modelo quantizado estão todos próximos o suficiente para serem indistinguíveis.

A saída do modelo final é um vetor muito longo, reproduz-se apenas um trecho que inclui o começo e o fim, para que se tenha uma ideia dos pesos obtidos no modelo.

```
unsigned char sine_model_quantized_tflite[] = {
0x1c, 0x00, 0x00, 0x00, 0x54, 0x46, 0x4c, 0x33, 0x00, 0x00, 0x12, 0x00,
0x1c, 0x00, 0x04, 0x00, 0x08, 0x00, 0x0c, 0x00, 0x10, 0x00, 0x14, 0x00,
// ...
0x00, 0x00, 0x08, 0x00, 0x0a, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x09,
};

unsigned int sine_model_quantized_tflite_len = 2512;
```

Para usar esse modelo em um projeto, você deve copiar e colar o conteúdo em um exemplo do TensorFlow Lite em seu microcontrolador. A Figura 29 apresenta a arquitetura da aplicação quando embarcada em um microcontrolador.

Figura 21 – Arquitetura final da aplicação do modelo em microcontrolador



Fonte: WARDEN, 2019, p. 50

4.3 Embarcando o modelo em um microcontrolador

Este modelo de código C pode ser embarcado num microcontrolador seguindo as instruções no TensorFlow Lite para Microcontroladores *hello_world/README.md*. Os modelos do Jupiter Notebook do hello_world podem ser encontrados no diretório *hello_world/train/models*.

Se você gerou um novo modelo, atualize os valores das variáveis definidas em *hello_world/model.cc* com os valores apresentadoes anteriormente como resultado da otimização.

Neste capítulo, vamos ligar e desligar uma sequencia de LED's utilizando o nosso modelo senoidal. Utilizando um laço contínuo, introduziremos um valor x no modelo, executaremos inferência, e depois utilizaremos o resultado para ligar e desligar um LED.

A fim de mostrar a menor implementação possível do TinyML, este programa em C++ 11 evita utilizar uma lógica complexa. Esta simplicidade o torna uma ferramenta útil para aprender TensorFlow Lite para Microcontroladores.

Iremos realizer um passo-a-passo do código de aplicação. Descrevendo como construí-lo e implementá-lo.

4.3.1 Incluindo Dependências

A diretriz *#include* especifica outro código no qual o código C++ se baseia. Aqui, nós usamos *#include* para importar os seguintes arquivos:

```
#include ‘‘tensorflow/lite/micro/examples/hello_world/sine_model_data.h
// O modelo seno treinado, convertido e transformado em C++ usando xxd
#include "tensorflow/lite/micro/kernels/all_ops_resolver.h"
// A classe que permite ao intérprete carregar as operações utilizadas
// pelo nosso modelo
#include "tensorflow/lite/micro/micro_error_reporter.h"
// A classe que pode registrar erros e saída para ajudar na depuração
#include "tensorflow/lite/micro/micro_interpreter.h"
// O TensorFlow Lite para microcontroladores intérprete, que executará
// nosso modelo
#include "tensorflow/lite/micro/testing/micro_test.h"
// A estrutura leve para testes de escrita, o que nos permite executar
// este arquivo como um teste
#include "tensorflow/lite/schema/schema_generated.h"
// O esquema que define a estrutura dos dados do TensorFlow Lite FlatBuffer,
// utilizado para dar sentido aos dados do modelo em sine_model_data.h
#include "tensorflow/lite/version.h"
// O número da versão atual do esquema, para que possamos verificar
// se o modelo foi definido com uma versão compatível
```

4.3.2 Mapeando o modelo

Usando o arquivo *sine_model_data.h*, podemos carregar os dados do modelo da seguinte forma:

```
// Mapear o modelo em uma estrutura de dados utilizável.
const tflite::Model* model = ::tflite::GetModel(g_sine_model_data);

if (model->version() != TFLITE_SCHEMA_VERSION) {
  error_reporter->Report(
    "Model provided is schema version %d not equal "
    "to supported version %d.\n",
    model->version(), TFLITE_SCHEMA_VERSION);
    return 1;
}
```

Passamos nossa matriz de dados do modelo (definida em sine_model_data.h) para um método chamado GetModel() na primeira linha. Um ponteiro Model é retornado, que é atribuído a uma variável chamada model. Esta variável representa nosso modelo, como você pode esperar.

Model é uma estrutura, que é muito semelhante a uma classe em C++. Ele é definido em *schema_generated.h*, e contém os dados de nosso modelo e nos permite consultá-lo.

4.3.3 Acesso às operações para TensorFlow

O próximo passo é criar uma instância de AllOpsResolver:

```
// Registra todas as implementações de operação que precisamos
tflite::ops::micro::AllOpsResolver resolver;
```

A classe *all_ops_resolver.h* define como o TensorFlow Lite para Microcontroladores acessa as operações. Usando AllOpsResolver, ele é capaz de fornecer todas as operações que estão disponíveis.

4.3.4 Alocação de memória

Como passo final, precisamos alocar uma área de memória de trabalho que nosso modelo irá utilizar:

```
// Criar uma área de memória a ser utilizada para entrada, saída
// e matrizes intermediárias.
// Encontrar o valor mínimo para seu modelo pode exigir alguma
// tentativa e erro.
const int tensor_arena_size = 2 × 1024;
uint8_t tensor_arena[tensor_arena_size];
```

Os modelos de entradas, saídas e tensores intermediários serão armazenados nesta área de memória. Nós a chamamos de tensor_arena. Em nosso caso, alocamos 2.048 bytes para a matriz. Neste caso, o especificamos como 2 × 1024.

4.3.5 Criação de intérpretes

Estamos prontos para montar o intérprete, agora que declaramos o tensor_arena:

```
// Construir um intérprete para executar o modelo
tflite::MicroInterpreter interpreter(model, resolver, tensor_arena,
                                     tensor_arena_size, error_reporter);

// Alocar a memória do tensor_arena para os tensores do modelo
interpreter.AllocateTensors();
```

Colocamos de lado uma área de memória, definindo uma matriz chamada tensor_arena. O método AllocateTensors() aloca a memória do tensor_arena a cada

um dos tensores definidos pelo modelo. AllocateTensors() deve ser chamado antes de executar a inferência, caso contrário, a inferência falhará. Tendo criado um intérprete, precisamos agora fornecer alguma informação para nosso modelo.

```
// Obter um ponteiro para o tensor de entrada do modelo
TfLiteTensor* input = interpreter.input(0);
```

O método de entrada input() do intérprete pode ser usado para definir um ponteiro para um tensor de entrada. Para especificar qual tensor de entrada queremos, devemos passar um índice para o método input(). Como temos apenas um tensor de entrada neste caso, o índice é 0.

4.3.6 Inferência de execução sobre uma entrada

A inferência é executada adicionando um valor ao nosso tensor de entrada e depois instruindo o intérprete a invocar o modelo. O modelo será então verificado para ver se foi executado com sucesso. O resultado é o seguinte:

```
// Fornecer um valor de entrada
input->data.f[0] = 0.;
// Execute o modelo nesta entrada e verifique se ele é bem sucedido
TfLiteStatus invoke_status = interpreter.Invoke();
if (invoke_status != kTfLiteOk) {
 error_reporter->Report("Invoke failed\n");
}
TF_LITE_MICRO_EXPECT_EQ(kTfLiteOk, invoke_status);''
```

A estrutura TfLiteTensor tem uma variável de dados que pode ser usada para definir o conteúdo do tensor de entrada. Ele é usado da seguinte forma:

```
input->data.f[0] = 0.;
```

É hora de fazer a inferência depois de configurar o tensor de entrada:

```
TfLiteStatus invoke_status = interpreter.Invoke();
```

4.3.7 Leitura da saída

Como na entrada, a saída de nosso modelo é acessada através de um TfLiteTensor, e obter um ponteiro para ele é igualmente simples.

```
TfLiteTensor* output = interpreter.output(0);
```

O tensor de saída será sobreescrito com novos valores toda vez que a inferência for feita. Portanto, se você quiser manter um valor de saída em seu programa durante a execução da inferência, você precisará copiá-lo do tensor de saída. Dessa forma, podemos ajustar a inferência de e executar contínuamente usando um superloop ou RTOS.

4.3.8 Entendendo o código

Até agora, realizamos tudo na nuvem através do Google Colab e do GitHub. Para executar nossos testes, devemos baixar e compilar o código em um computador. Para isso, precisamos das seguintes ferramentas de software:

1. Um emulador de terminal, como o Terminal do macOS.
2. Um shell bash (o padrão do macOS antes do Catalina e da maioria das distribuições Linux).
3. Git (instalado por padrão no macOS e na maioria das distribuições Linux).
4. Make, versão 3.82 ou posterior.

Uma vez que você tenha todas as ferramentas, abra um terminal e digite o comando abaixo para baixar o código fonte do TensorFlow, que inclui o código de exemplo que estamos usando. Se você executá-lo no seu diretorio atual, ele criará um diretório contendo o código fonte:

```
git clone https://github.com/tensorflow/tensorflow.git
```

Mudando para o diretório do tensor recém-criado:

```
cd tensorflow
```

Os desenvolvedores de software utilizam o Make para automatizar as tarefas de construção (build). Ele e utsado desde 1976, o que em termos de computação é quase para sempre. Para construir e executar códigos, os desenvolvedores escrevem arquivos chamados Makefile que instruem o Make a fazer isso. Existe um Makefile definido em *micro/tools/make/Makefile* para TensorFlow Lite para Microcontroladores. Cobriremos como usar o Make nas seções seguintes.

4.3.9 Estrutura de arquivo de um projeto

A raiz da aplicação está em *tensorflow/lite/micro/exemplos/hello*. Ela contém os seguintes arquivos:

- *BUILD*: Lista as várias coisas que podem ser construídas usando o código fonte da aplicação, incluindo o binário da aplicação principal e os testes. Não precisamos nos preocupar muito com isso neste momento.
- *Makefile.inc*: Este Makefile contém informações sobre os dispositivos para a construção de nossa aplicação, incluindo o "hello world"e seu binário principal. Ele define quais arquivos fonte são parte deles.
- *README.md*: Um arquivo de leitura contendo instruções sobre como construir e executar o aplicativo.
- *constants.h e constantes.cc*: Um par de arquivos contendo várias constantes (variáveis que não mudam durante a vida útil de um programa) que são importantes para definir o comportamento do programa.
- *criate_sine_model.ipynb*: O Jupyter notebook usado para gerar o modelo.
- *hello_world_test.cc*: Um teste que faz a inferência usando nosso modelo.
- *main.cc*: O ponto de entrada do programa, que roda primeiro quando a aplicação é embarcada em um dispositivo.
- *main_functions.h e main_functions.cc*: Um par de arquivos que definem uma função setup(), que executa toda a inicialização requerida por nosso programa, e uma função loop(), que contém a lógica central do programa e é projetada para ser chamada repetidamente em um loop. Estas funções são chamadas pelo main.cc quando o programa é iniciado.
- *output_handler.h e output_handler.cc*: Um par de arquivos que definem uma função que podemos usar para exibir uma saída cada vez que a inferência é executada. A implementação padrão, no output_handler.cc, imprime o resultado para a tela. Podemos sobrepor esta implementação para que ela faça coisas diferentes em dispositivos diferentes.
- *output_handler_test.cc*: Um teste que prova que o código em output_handler.h e output_handler.cc está funcionando corretamente.
- *sine_model_data.h e sine_model_data.cc*: Um par de arquivos que definem uma matriz de dados representando nosso modelo, como exportado usando xxd na primeira parte deste capítulo.

Este diretório também contém estes subdiretórios (e talvez mais):

```
arduino/
disco_f76ng/
sparkfun_edge/
```

Devido às diferentes capacidades e APIs de diferentes plataformas de microcontroladores, esta estrutura de projeto nos permite fornecer versões específicas do dispositivo de arquivos fonte que serão utilizados em vez dos padrões, se a aplicação for construída para aquele dispositivo. Por exemplo, o diretório arduino contém versões específicas para o Arduino como o main.cc, constants.cc, e output_handler.cc.

4.3.10 Começando com o main_functions.cc

O arquivo declara uma função chamada setup(). Ela será chamada uma vez no início do programa, mas nunca mais depois disso. Antes de executar as inferências, nós o usamos para lidar com todo o trabalho doméstico ocasional.

O logging é montado, o modelo é carregado, o intérprete é montado e a memória é alocada:

```
void setup() {
  // Logging.
  static tflite::MicroErrorReporter micro_error_reporter;
  error_reporter = &micro_error_reporter;
  // Mapear o modelo em uma estrutura de dados utilizável.
  model = tflite::GetModel(g_sine_model_data);
  if (model->version() != TFLITE_SCHEMA_VERSION) {
    error_reporter->Report(
        "Model provided is schema version %d not equal "
        "to supported version %d.",
        model->version(), TFLITE_SCHEMA_VERSION);
    return;
  }
  // Isto atrai todas as implementações de operação que precisamos.
  static tflite::ops::micro::AllOpsResolver resolver;
  // Construir um intérprete para executar o modelo com.
  static tflite::MicroInterpreter static_interpreter(
      model, resolver, tensor_arena, kTensorArenaSize, error_reporter);
  interpreter = &static_interpreter;
  // Alocar memória do tensor_arena para os tensores do modelo.
  TfLiteStatus allocate_status = interpreter->AllocateTensors();
  if (allocate_status != kTfLiteOk) {
    error_reporter->Report("AllocateTensors() failed");
    return;
  }
```

Agora montamos toda a nossa infra-estrutura de aprendizagem de máquinas. A inferência pode ser executada e os resultados podem ser obtidos usando todas as ferramentas fornecidas. A seguir, precisamos definir nossa lógica de aplicação.

O modelo foi treinado para prever o seno de qualquer número entre 0 e 2π, o que representa o ciclo completo de uma onda senoidal. A fim de demonstrar nosso

modelo, poderíamos inserir números nesta faixa, prever seus pecados e, em seguida, emitir os valores. Isto poderia ser feito em uma seqüência para mostrar como o modelo funciona em toda a faixa.

Precisamos escrever algum código de looping para realizar isto. Nosso próximo passo será declarar uma função chamada loop(). Esta função executará repetidamente nossa lógica de aplicação.

```
void loop()
```

A primeira coisa que devemos fazer em nossa função loop() é determinar que valor passar para o modelo (vamos chamá-lo de valor x). Determinamos isto usando duas constantes: kXrange, que especifica o máximo valor x possível como 2, e kInferencesPerCycle, que define o número de inferências que queremos realizar ao passar de 0 para 2π. Calculamos o valor x da seguinte forma:

```
// Calcule um valor x para alimentar o modelo. Comparamos o valor atual
// inference_count ao número de inferências por ciclo para determinar
// nossa posição dentro da faixa de possíveis x valores o modelo era
// treinados, e use isto para calcular um valor.
float position = static_cast<float>(inference_count) /
                     static_cast<float>(kInferencesPerCycle);
float x_val = position * kXrange;
```

Nossa posição atual dentro da faixa é determinada pela divisão das inferências (que é o número de inferências que fizemos até agora) por kInferencesPerCycle. Este valor é multiplicado por kXrange, que representa o valor máximo no intervalo (2π). Este valor, x_val, será passado para o nosso modelo.

Usamos duas constantes, kInferencesPerCycle e kXrange, definidas em constants.h e constantes.cc. As constantes são prefixadas com um k em C++, facilitando a sua identificação como constantes. Se as constantes forem definidas em um arquivo separado, elas podem ser incluídas e usadas em qualquer lugar onde forem necessárias.

A próxima parte de nosso código deve parecer agradável e familiar; escrevemos nosso valor x no tensor de entrada do modelo, fazemos a inferência, e então pegamos o resultado (vamos chamá-lo de nosso valor y) do tensor de saída:

```
// Colocar nosso valor x calculado no tensor de entrada do modelo
input->data.f[0] = x_val;
// Executar inferência, e relatar qualquer erro
TfLiteStatus invoke_status = interpreter->Invoke();
if (invoke_status != kTfLiteOk) {
  error_reporter->Report("Invoke failed on x_val: %f\n",
```

```
                          static_cast<double>(x_val));
  return;
}
// Leia o valor previsto y do tensor de saída do modelo
float y_val = output->data.f[0];
```

O valor senoidal foi agora determinado. O processo de inferência leva um pouco de tempo, e este código corre em loop, portanto, vamos gerar valores senoidais ao longo do tempo. Uma animação ou LEDs piscantes podem ser controlados com isto. O próximo passo é produzi-lo de alguma forma.

Na saída_handler.cc, a função HandleOutput() é chamada de HandleOutput():

```
// Produzir os resultados. Uma função HandleOutput personalizada
// pode ser implementada para cada hardware suportado.
HandleOutput(error_reporter, x_val, y_val);
```

Nossos valores x e y são passados junto com nossa instância ErrorReporter, que podemos usar para registrar erros.

4.3.11 Entendendo o main.cc

Todo programa C++ deve conter uma função global denominada main(), que é executada quando o programa é iniciado. A função main() é definida no arquivo main.cc em nosso programa. Ter esta função main() é o que faz do main.cc o ponto de entrada para nosso programa. Sempre que o microcontrolador inicia, main() executa o código.

Há apenas algumas linhas no main.cc. Primeiro, ele contém uma declaração #include para main_functions.h, que inclui setup() e loop():

```
#include "tensorflow/lite/micro/examples/hello_world/main_functions.h"
// Em seguida, declara a função main() propriamente dita:
int main(int argc, char* argv[]) {
  setup();
  while (true) {
    loop();
  }
}
```

A função main() chama primeiro a setup(). Isto será feito apenas uma vez. Em seguida, entra um loop que repetidamente chama a função loop(). Ela continuará a funcionar indefinidamente.

4.3.12 Executando nossa aplicação

O primeiro passo para testar nosso código de aplicação é construir um binário. Para criar um binário executável, digite o seguinte comando:

```
make -f tensorflow/lite/micro/tools/make/Makefile hello_world
```

Após a conclusão da construção, você pode executar o binário da aplicação usando o seguinte comando, dependendo de seu sistema operacional:

```
# macOS:
tensorflow/lite/micro/tools/make/gen/osx_x86_64/bin/hello_world

# Linux:
tensorflow/lite/micro/tools/make/gen/linux_x86_64/bin/hello_world

# Windows
tensorflow/lite/micro/tools/make/gen/windows_x86_64/bin/hello_world
```

Tente listar os diretórios em *tensorflow/lite/micro/ferramentas/make/gen/* se você não conseguir encontrar o caminho correto. Você deve ver algo como isto depois de executar o binário:

```
x_value: 1.4137159*2^1, y_value: 1.374213*2^-2
x_value: 1.5707957*2^1, y_value: -1.4249528*2^-5
x_value: 1.7278753*2^1, y_value: -1.4295994*2^-2
x_value: 1.8849551*2^1, y_value: -1.2867725*2^-1
x_value: 1.210171*2^2, y_value: -1.7542461*2^-1
```

HandleOutput() do output_handler.cc escreve estes logs. Cada inferência é representada por um log, onde o valor x aumenta lentamente até chegar a 2, ponto em que volta a 0 e começa de novo. Você pode terminar o programa pressionando Ctrl-C quando terminar.

4.3.13 Gravando o programa na Sparkfun Edge

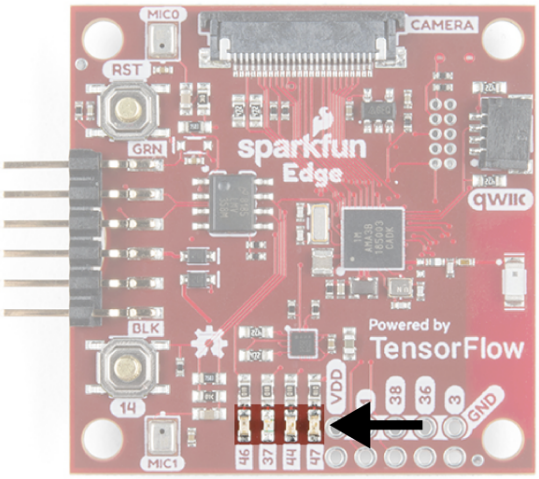
Um microcontrolador vive em uma placa de circuito conectado por pinos. Um microcontrolador geralmente tem dezenas de pinos que servem para várias finalidades. Os microcontroladores são alimentados por alguns; outros são conectados a vários componentes. Os sinais digitais são entrados e saídos através de alguns pinos no microcontrolador por programas.

Estes pinos são conhecidos como pinos GPIO, que significa entrada/saída para fins gerais. É possível que eles funcionem como entradas, determinando se uma

tensão é aplicada a eles, ou como saídas, fornecendo corrente para alimentação ou comunicação com outros dispositivos.

A SparkFun Edge foi especificamente projetada para experimentar a aprendizagem da máquina em dispositivos embarcados. Este dispositivo utiliza um microcontrolador Ambiq Apollo 3 com um núcleo processador ARM Cortex M4.

Figura 22 – LED's da SparkFun



Fonte: Do próprio autor.

Usando o banco de LEDs da placa, podemos criar uma animação simples. Esta é a ordem em que os LEDs estão fisicamente dispostos: vermelho, verde, azul e amarelo. Para valores y diferentes, acenderemos os LEDs da seguinte forma:

```
Range              |  LEDs lit

0.75 <= y <= 1.    |  [ 0 0 1 1 ]
0 < y < 0.75.      |  [ 0 0 1 0 ]
y = 0.             |  [ 0 0 0 0 ]
-0.75 < y < 0      |  [ 0 1 0 0 ]
-1 <= y <= 0.75    |  [ 1 1 0 0 ]
```

O kInferencesPerCycle, definido em constants.cc, ajusta a velocidade do ciclo de LEDs com base no número de inferências por ciclo.

A primeira coisa que precisamos fazer é incluir alguns arquivos de cabeçalho. A interface deste arquivo é definida em output_handler.h. Outro arquivo, am_bsp.h, vem do Ambiq Apollo3 SDK. A Ambiq fabrica o microcontrolador da SparkFun Edge, o Apollo3. Os microcontroladores são controlados por kits de desenvolvimento de software (SDKs), que contêm arquivos fonte que definem as constantes e funções.

```
void HandleOutput(tflite::ErrorReporter* error_reporter, float x_value,
                  float y_value) {
```

```
  // A primeira vez que este método funciona
  // configure nossos LEDs corretamente
  static bool is_initialized = false;
  if (!is_initialized) {
    // Set up LEDs as outputs
    am_hal_gpio_pinconfig(AM_BSP_GPIO_LED_RED, g_AM_HAL_GPIO_OUTPUT_12);
    am_hal_gpio_pinconfig(AM_BSP_GPIO_LED_BLUE, g_AM_HAL_GPIO_OUTPUT_12);
    am_hal_gpio_pinconfig(AM_BSP_GPIO_LED_GREEN, g_AM_HAL_GPIO_OUTPUT_12);
    am_hal_gpio_pinconfig(AM_BSP_GPIO_LED_YELLOW, g_AM_HAL_GPIO_OUTPUT_12);
    // Ensure all pins are cleared
    am_hal_gpio_output_clear(AM_BSP_GPIO_LED_RED);
    am_hal_gpio_output_clear(AM_BSP_GPIO_LED_BLUE);
    am_hal_gpio_output_clear(AM_BSP_GPIO_LED_GREEN);
    am_hal_gpio_output_clear(AM_BSP_GPIO_LED_YELLOW);
    is_initialized = true;
  }
```

A função am_hal_gpio_pinconfig() no am_bsp.h é usada para colocar os pinos que se conectam aos LEDs da placa no modo de saída (representado por g_AM_HAL_GPIO_OUTPUT_12). Os números dos pinos LED são representados por constantes, tais como AM_BSP_GPIO_LED_RED.

No próximo passo, limpamos todas as saídas, para que os LEDs estejam todos desligados. Quando o valor y é negativo, determinamos quais LEDs devem ser acesos:

```
// Configurar os LEDs para representar valores negativos
if (y_value < 0) {
  // Limpar LEDs desnecessários
  am_hal_gpio_output_clear(AM_BSP_GPIO_LED_GREEN);
  am_hal_gpio_output_clear(AM_BSP_GPIO_LED_YELLOW);
  // O LED azul está aceso para todos os valores negativos
  am_hal_gpio_output_set(AM_BSP_GPIO_LED_BLUE);
  // O LED vermelho está aceso apenas em alguns casos
  if (y_value <= -0.75) {
    am_hal_gpio_output_set(AM_BSP_GPIO_LED_RED);
  } else {
    am_hal_gpio_output_clear(AM_BSP_GPIO_LED_RED);
  }
```

Se o valor y apenas tornou-se negativo, limpamos os dois LEDs que indicam valores positivos. Em seguida, chamamos am_hal_gpio_output_set() para ligar o LED azul, que está sempre aceso se o valor for negativo. Finalmente, se o valor for inferior a -0,75, o LED vermelho é ligado. Caso contrário, ele é desligado.

Para valores positivos de y, fazemos a mesma coisa:

```
// Configurar os LEDs para representar valores positivos
} else if (y_value > 0) {
  // Limpar LEDs desnecessários
  am_hal_gpio_output_clear(AM_BSP_GPIO_LED_RED);
  am_hal_gpio_output_clear(AM_BSP_GPIO_LED_BLUE);
  // O LED verde está aceso para todos os valores positivos
  am_hal_gpio_output_set(AM_BSP_GPIO_LED_GREEN);
  // O LED amarelo está aceso apenas em alguns casos
  if (y_value >= 0.75) {
    am_hal_gpio_output_set(AM_BSP_GPIO_LED_YELLOW);
  } else {
    am_hal_gpio_output_clear(AM_BSP_GPIO_LED_YELLOW);
  }
}
```

Isso é tudo para os LEDs. Finalmente, registramos os valores de saída atuais na porta serial para qualquer dispositivo que esteja escutando:

```
// Registrar os valores X e Y atuais
error_reporter->Report("x_value: %f, y_value: %f\n", x_value, y_value);
```

O processo de construção pode ter mudado desde que este livro foi escrito, então verifique README.md para obter as últimas instruções.

Para compilar um binário SparkFun Edge, faça o download de todas as dependências necessárias e execute o seguinte comando:

```
make -f tensorflow/lite/micro/tools/make/Makefile \
  TARGET=sparkfun_edge hello_world_bin
```

Um arquivo binário contém o programa para que o hardware SparkFun Edge possa executá-lo diretamente. Um arquivo .bin será criado no seguinte local:

```
tensorflow/lite/micro/tools/make/gen/ \
  sparkfun_edge_cortex-m4/bin/hello_world.bin
```

A assinatura do binário com chaves criptográficas é necessária antes da implantação. Vamos agora executar alguns comandos no SparkFun Edge para assinar o binário. Usamos os scripts Ambiq SDK, que são baixados quando o Makefile é executado.

Usando o seguinte comando, você pode gerar algumas chaves criptográficas fictícias para usar durante o desenvolvimento:

```
cp tensorflow/lite/micro/tools/make/downloads/AmbiqSuite-Rel2.0.0/ \
  tools/apollo3_scripts/keys_info0.py \
  tensorflow/lite/micro/tools/make/downloads/AmbiqSuite-Rel2.0.0/ \
  tools/apollo3_scripts/keys_info.py
```

A seguir, execute o seguinte comando para criar um binário assinado:

```
python3 tensorflow/lite/micro/tools/make/downloads/ \
  AmbiqSuite-Rel2.0.0/tools/apollo3_scripts/create_cust_image_blob.py \
  --bin tensorflow/lite/micro/tools/make/gen/ \
  sparkfun_edge_cortex-m4/bin/hello_world.bin \
  --load-address 0xC000 \
  --magic-num 0xCB -o main_nonsecure_ota \
  --version 0x0
```

Isto cria o arquivo binário main_nonsecure_ota.bin. Execute o seguinte comando para criar uma versão final do arquivo que você usará no próximo passo para piscar seu dispositivo:

```
python3 tensorflow/lite/micro/tools/make/downloads/ \
  AmbiqSuite-Rel2.0.0/tools/apollo3_scripts/create_cust_wireupdate_blob.py \
  --load-address 0x20000 \
  --bin main_nonsecure_ota.bin \
  -i 6 \
  -o main_nonsecure_wire \
  --options 0x1
```

O diretório onde você executou os comandos deve conter agora um arquivo chamado main_nonsecure_wire.bin. Você irá flashar este arquivo para o dispositivo.

O programador serial Serial Básico SparkFun USB-C é usado para baixar novos programas para a placa. A interface USB permite que seu computador se comunique com o microcontrolador. Siga estes passos para anexar este dispositivo a sua placa:

- Na lateral da borda da SparkFun, localize o cabeçalho de seis pinos.
- Conecte o SparkFun USB-C Serial Basic a estes pinos, alinhando os pinos rotulados BLK e GRN em cada dispositivo.

Uma vez que a placa tenha sido conectada ao seu computador via USB, você pode começar a usá-la. A placa deve ser nomeada por seu computador para que possa ser programada. Você pode descobrir qual dispositivo é novo listando todos os dispositivos do computador antes e depois de anexá-lo.

É altamente recomendável que você instale o driver antes de continuar, já que algumas pessoas relataram problemas com os drivers padrão de seu sistema operacional. Antes de conectar o dispositivo via USB, execute o seguinte comando:

```
# macOS:
ls /dev/cu*

# Linux:
ls /dev/tty*
```

Isto deve produzir uma lista de dispositivos como:

```
/dev/cu.Bluetooth-Incoming-Port
/dev/cu.MALS
/dev/cu.SOC
```

Agora, conecte o programador à porta USB do seu computador e execute o comando novamente:

```
# macOS:
ls /dev/cu*

# Linux:
ls /dev/tty*
```

A saída deve incluir um item adicional, como mostrado no exemplo a seguir. É possível que seu novo item tenha um nome diferente. O dispositivo é nomeado da seguinte forma:

```
/dev/cu.Bluetooth-Incoming-Port
/dev/cu.MALS
/dev/cu.SOC
/dev/cu.wchusbserial-1450
```

Sempre que fizermos referência a este dispositivo, usaremos este nome. Ele pode, entretanto, mudar dependendo da porta USB em que o programador está conectado, portanto, se você desconectar e recolocar a placa, talvez seja necessário verificar seu nome novamente. Identifique o nome do dispositivo e guarde-o em uma variável de ambiente:

```
export DEVICENAME=<your device name here>''
```

Agora, configuramos uma variável de ambiente para especificar o baud rate, que é a velocidade na qual os dados serão enviados para o dispositivo:

```
export BAUD_RATE=921600
```

Copie o seguinte comando e cole-o em seu terminal. Não pressione enter ainda. Você precisará substituir DEVICENAME e BAUD_RATE no comando pelos valores que você definiu nas seções anteriores:

```
python3 tensorflow/lite/micro/tools/make/downloads/ \
  AmbiqSuite-Rel2.0.0/tools/apollo3_scripts/ \
  uart_wired_update.py -b ${BAUD_RATE} \
  ${DEVICENAME} -r 1 -f main_nonsecure_wire.bin -i 6
```

Ao desligar e ligar a placa, você reiniciará em seu estado de bootloader. Agora você deve ver algo como o seguinte na tela:

```
Connecting with Corvette over serial port /dev/cu.usbserial-1440...
Sending Hello.
Received response for Hello
Received Status
length =  0x58
version =  0x3
Max Storage =  0x4ffa0
Status =  0x2
State =  0x7
AMInfo =
0x1
0xff2da3ff
0x55fff
0x1
0x49f40003
0xffffffff
...
0xffffffff
Sending OTA Descriptor =  0xfe000
Sending Update Command.
number of updates needed =  1
Sending block of size  0x158b0  from  0x0  to  0x158b0
Sending Data Packet of length  8180
Sending Data Packet of length  8180
...
```

Segure o botão 14 da placa até ver a mensagem *Sending Data Packet of length 8180*. Uma vez que você veja isto, você pode liberar o botão. As linhas continuarão a ser impressas no terminal pelo programa. Você eventualmente verá algo como:

```
...
Sending Data Packet of length  8180
Sending Data Packet of length  6440
Sending Reset Command.
Done.
```

Quando isto acontece a gravação foi bem sucedida. Agora você poderá separar sua placa do USB e ver o LED piscando com a sequência que determinamos para eles anteriormente. A aplicação deve estar sendo executada conforme o esperado.

4.4 Caso de uso: detecção de palavras com microfone

Este caso de uso utiliza o microfone *onboard*, para a detecção de palavras e também informa se uma palavra não foi reconhecida. O mesmo princípio desta aplicação pode ser aplicado a qualquer sensor tal como um acelerômetro, um sensor de temperatura e outros sensores, muito comuns em aplicações mecatrônicas.

Este exemplo foi aplicado apenas na plataforma Spark Fun, e permitiu o reconhecimento das palavras “sim”, “não”, assim como “não reconhecido”, acendendo os LED’s *onboard* da placa. O LED amarelo foi utilizado para representar a palavra “sim”, já o LED vermelho para a palavra “não” e finalmente o LED verde para quando não foi possível interpretar a palavra.

Esta aplicação utiliza os conceitos de Janelamento, Transformada Rápida de Fourier (FFT) e outros, que tornam mais complexa a aquisição de dados, porém são necessárias pois a taxa de aquisição é muito elevada. O Apêndice 2 apresenta um passo a passo de como executar o mesmo projeto, assim como o procedimento para gerar o seu código binário embarcável.

4.4.1 Importância de se executar inferência em palavras localmente

Na maioria dos casos, o trabalho pesado de reconhecimento de fala, processamento de linguagem natural e geração de respostas às consultas dos usuários é feito na nuvem computacional, em servidores poderosos que executam grandes modelos de aprendizado de máquina. Quando um usuário faz uma pergunta, ela é enviada ao servidor como um fluxo de áudio. O servidor descobre o que isso significa, procura todas as informações necessárias e envia a resposta apropriada de volta.

Mas parte do apelo de um assistente por voz é que eles estão sempre ligados, prontos para ajudá-lo. Ao dizer "Hey Google"ou "Alexa", você pode acordar seu assistente e dizer o que precisa sem precisar pressionar um botão. Isso significa que eles devem estar ouvindo sua voz 24 horas por dia, 7 dias por semana, esteja você sentado em sua sala de estar, dirigindo pela rodovia ou ao ar livre com um telefone na mão. o uso da tecnologia *Wake word* (literalmente "palavra que acorda") é o ideal para sistemas que utilizam TFLM.

Um caso de uso poderia ser a identificação de uma palavra, por exemplo, "socorro", para acionar automaticamente uma parada de emergência em máquinas, ou mesmo ligar ou desligar processos.

4.4.2 Introdução ao modelo a ser gerado

O modelo foi treinado com um acervo de dados chamado de *conjunto de dados de comandos de fala* (TENSORFLOW, 2022d). Isso consiste em 65.000 declarações de um segundo de duração de 30 palavras curtas. Estes dados podem ser encontrados online e foram adquiridos por meio de um crowdsource online.

Embora o conjunto de dados contenha 30 palavras diferentes, o modelo foi treinado para distinguir entre apenas quatro categorias: as palavras "yes"e "no", palavras "desconhecidas"(ou seja, as outras 28 palavras no conjunto de dados) e silêncio.

O modelo não recebe dados brutos de amostra de áudio. Em vez disso, ele funciona com espectrogramas, que são matrizes bidimensionais que são compostas de fatias de informações de frequência, cada uma tirada de uma janela de tempo diferente.

Ao isolar as informações de frequência durante o pré-processamento, facilita-se a geração do modelo. Durante o treinamento, ele não precisa aprender a interpretar dados brutos de áudio; em vez disso, ele começa a trabalhar com uma abstração de camada superior que destila as informações mais úteis.

Há um tipo de arquitetura de rede neural que é projetada especificamente para funcionar bem com tensores multidimensionais nos quais a informação está contida nas relações entre grupos de valores adjacentes. É chamado de rede neural convolucional (CNN).

O exemplo mais comum desses tipos de dados são as imagens, para as quais um grupo de *pixels* adjacentes pode representar uma forma, padrão ou textura. Durante o treinamento, uma CNN é capaz de identificar essas características e aprender o que elas representam. O modelo pode aprender como recursos de imagem simples (como linhas ou bordas) se encaixam em recursos mais complexos (como um olho ou uma orelha).

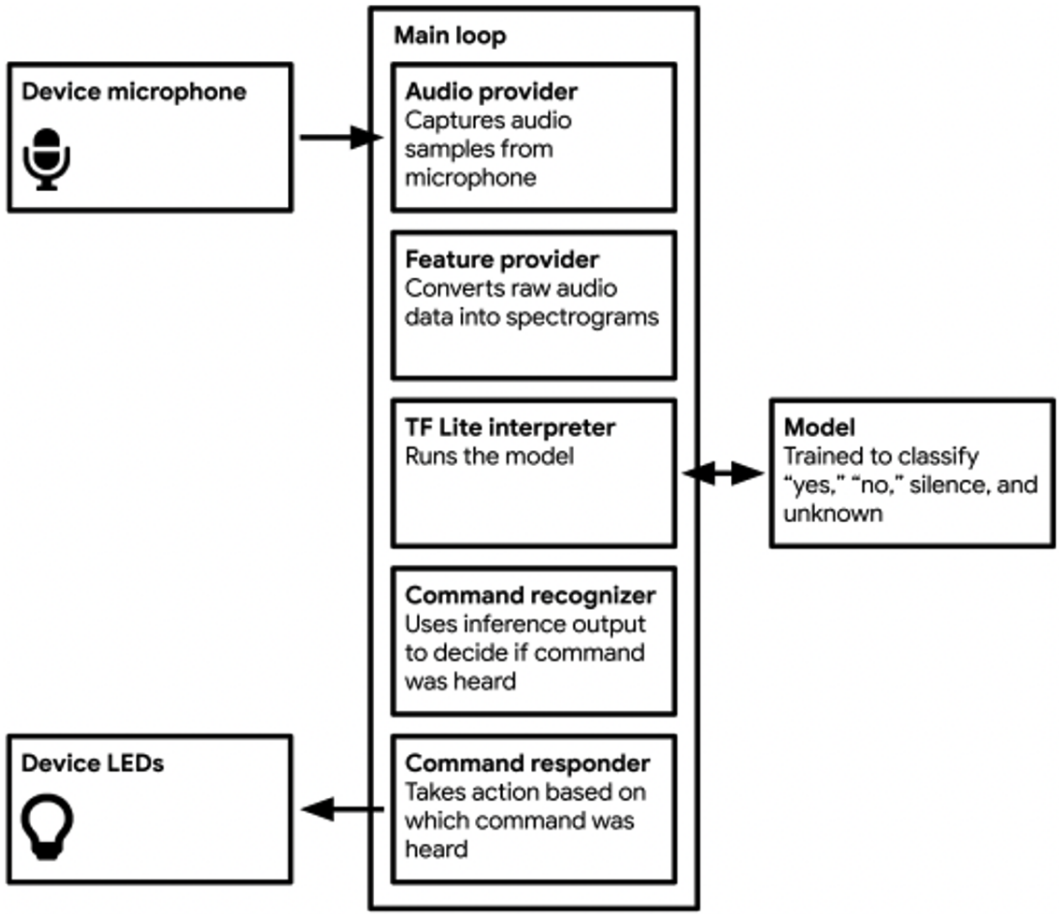
4.4.3 Estrutura da aplicação

A Figura 30 mostra uma visão geral do software. Este software é executado em um laço contínuo e todos os processos subsequentes estão contidos nele. O provedor de áudio captura dados de áudio brutos do microfone. O provedor de recursos converte dados brutos de áudio no formato de espectrograma que nosso modelo exige. O interpretador executa o modelo TensorFlow Lite, transformando o espectrograma de entrada em um conjunto de probabilidades.

O modelo é incluído como uma matriz de dados e executado pelo interpretador. Como a inferência é executada várias vezes por segundo, o registro de comandos agrega os resultados e determina se, em média, uma palavra conhecida foi ouvida.

Se um comando foi ouvido, o respondente de comando usa os recursos de saída do dispositivo para informar o usuário.

Figura 23 – Exemplo de aplicação usando TensorFlow Lite com microfone



Fonte: (WARDEN, 2019, p. 93)

4.4.4 Fluxo da aplicação

O exemplo codificado em C pelo autor utiliza o mesmo fluxo básico do exemplo anterior para a função seno no microcontrolador. A maior diferença está nos periféricos que utilizamos da plataforma de desenvolvimento escolhida e dos espectogramas que são nossas entradas.

Como estamos lidando com um espectrograma como nossa entrada, o tensor de entrada tem mais dimensões. A primeira dimensão é apenas um invólucro contendo um único elemento. O segundo e o terceiro representam as *linhas* e *colunas* do nosso espectrograma. A quarta dimensão mais interna do tensor de entrada, que tem tamanho 1, contém cada *pixel* individual do espectrograma.

Nossa produção tem duas dimensões. O primeiro é apenas um invólucro. O segundo tem quatro elementos. Esta é a estrutura que contém as probabilidades de que cada uma de nossas quatro classes (silêncio, desconhecido, "yes"e "no") foram correspondidas.

Passamos em um espectrograma "yes", então esperamos que a variável *yes_score* contenha uma probabilidade maior do que *silence_score*, *unknown_score* e *no_score*. Quando estamos satisfeitos com o "yes", fazemos a mesma coisa com um espectrograma "no". Primeiro, copiamos em uma entrada e executamos a inferência.

4.4.5 Capturando áudio

Cada dispositivo tem um mecanismo diferente para capturar áudio. Conforme descrito em audio_provider.h, espera-se que a função retorne uma matriz de dados de áudio modulados por código de pulso (PCM) de 16 bits. Este é um formato muito comum para áudio digital. O provedor de áudio apresentado no código-fonte é usado pelo provedor de recursos como uma fonte de amostras de áudio novas, fornecendo ao código principal ponteiros para seu conteúdo.

4.4.6 Convertendo áudio em espectogramas

O provedor de recursos converte áudio bruto, obtido do provedor de áudio, em espectrogramas que podem ser alimentados em nosso modelo. Ele é chamado durante o *loop* principal.

Seu trabalho é preencher uma matriz que representa um espectrograma de um segundo de áudio. Para evitar trabalho desnecessário, ele gerará novos recursos apenas para o tempo entre agora e quando foi chamado pela última vez. Se ele fosse chamado há menos de um segundo, ele manteria parte de sua saída anterior e geraria apenas as partes ausentes.

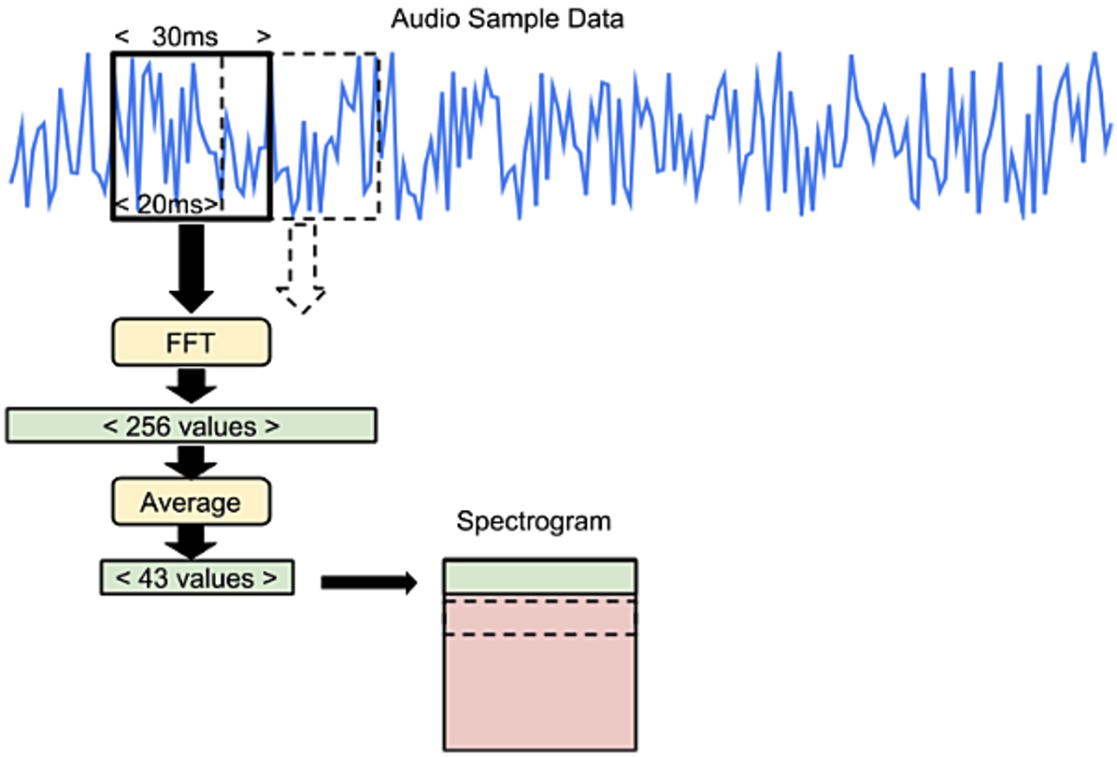
No código-fonte fornecido pelo autor, cada espectrograma é representado como uma matriz 2D, com 40 colunas e 49 linhas, onde cada linha representa uma amostra de áudio de 30 milissegundos (ms) dividida em 43 *frequency bins* (ou *frequency buckets*).

Para criar cada linha, executa-se uma fatia de 30 ms de entrada de áudio através de um algoritmo de Transformada Rápida de Fourier (FFT). Essa técnica analisa a distribuição de frequência do áudio na amostra e cria uma matriz de 256 "buckets"de frequência, cada um com um valor de 0 a 255. Estes são calculados em média em grupos de seis, deixando-nos com 43 buckets.

Os *frequency bins* são intervalos entre amostras no domínio da frequência. Por exemplo, se a taxa de amostragem for 100 Hz e o tamanho da FFT for 100, você terá 100 pontos entre 0 e 100 Hz. Portanto, você divide toda a faixa de 100 Hz em 100 intervalos, como 0-1 Hz, 1-2 Hz e assim por diante. Cada um desses pequenos intervalos, tal como o de 0-1 Hz neste exemplo, é um *frenquecy bin* ou *frequency bucket*.

Para construir toda a matriz 2D, combinamos os resultados da execução do FFT em 49 fatias consecutivas de áudio de 30 ms, com cada fatia sobrepondo a última em 10 ms. A figura 31 mostra como isso acontece.

Figura 24 – Janelamento para criação de espectrograma



Fonte: (WARDEN, 2019, p. 100)

A janela de amostra de 30 ms é movida para a frente em 20 ms de cada vez até que tenha coberto a amostra completa de um segundo. O espectrograma resultante está pronto para passar para o nosso modelo.

Depois de mover os dados, ele inicia um laço que itera uma vez para cada nova fatia necessária. Nesse laço, ele primeiro solicita áudio para essa fatia do provedor de áudio que está usando.

Se você estiver criando seu próprio aplicativo que usa espectrogramas, poderá reutilizar esse código como está. Você precisará usar o mesmo código para pré-processar dados em espectrogramas ao treinar seu modelo.

Uma vez que temos um espectrograma, podemos executar inferência sobre ele usando o modelo. Depois que isso acontece, precisamos interpretar os resultados.

4.4.7 Reconhecendo comandos

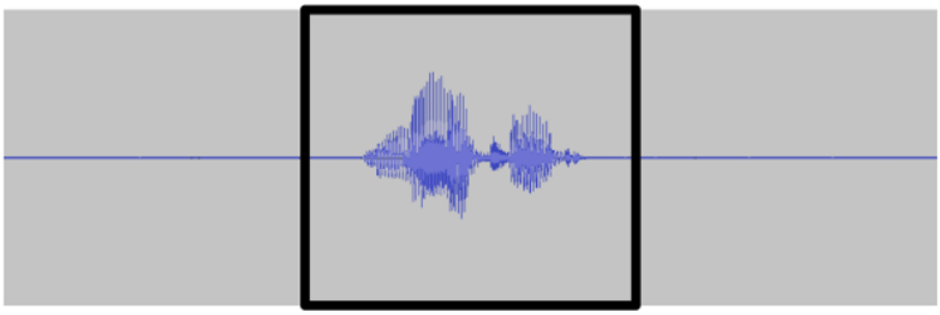
Depois que nosso modelo produz um conjunto de probabilidades de que uma palavra conhecida foi falada no último segundo de áudio, precisamos reconhecê-las como comandos. Parece que isso seria simples: se a probabilidade em uma

determinada categoria é maior do que um certo limiar, a palavra foi falada. No entanto, no mundo real, as coisas se tornam um pouco mais complicadas.

Ele está executando várias inferências por segundo, cada uma em uma janela de um segundo de dados. Isso significa que executaremos inferência em qualquer palavra várias vezes, em várias janelas, dependendo do tamanho da janela, podemos estar perdendo o espectrograma adequado.

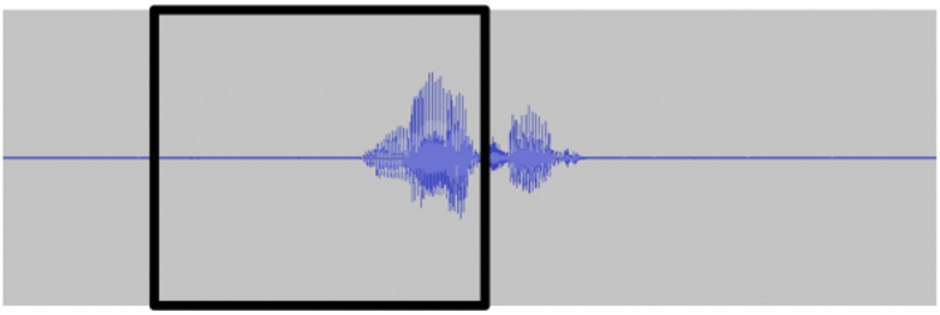
Na Figura 29, pode-se ver uma forma de onda da palavra "noted"sendo falada, cercada por uma caixa que representa uma janela de um segundo sendo capturada. Nosso modelo é treinado para detectar a palavra "no"e entende que a palavra "noted"não é a mesma coisa. Se executarmos a inferência nessa janela de um segundo, ela produzirá uma baixa probabilidade para a palavra "no".

Figura 25 – A palavra noted sendo capturada na janela



Fonte: (WARDEN, 2019, p. 102)

Figura 26 – A palavra noted sendo capturada parcialmente na janela



Fonte: (WARDEN, 2019, p. 102)

No entanto, e se a janela viesse um pouco mais cedo no fluxo de áudio, como na Figura 30, a única parte da palavra "noted"que aparece dentro da janela é a sua primeira sílaba. Como a primeira sílaba de "noted"soa como "no", é provável que o modelo interprete isso como tendo uma alta probabilidade de ser um "no".

Esse problema, juntamente com outros, significa que não podemos confiar em uma única inferência para nos dizer se uma palavra foi dita. O reconhecedor calcula a pontuação média de cada palavra nas últimas inferências e decide se ela é alta o

suficiente para contar como uma detecção. Para fazer isso, alimentamos cada resultado de inferência à medida que eles acontecem.

4.4.8 Respondendo a comandos

A peça final em nosso quebra-cabeça, o comando responder, é o que produz uma saída para nos informar que uma palavra foi detectada.

O respondente de comando foi projetado para ser substituído para cada tipo de dispositivo. Explorar-se-á as implementações específicas do dispositivo mais adiante neste capítulo. Por enquanto, vamos dar uma olhada em sua implementação de referência muito simples, que apenas registra os resultados da detecção como texto.

4.4.9 Executando a aplicação

O número após cada palavra detectada é a sua pontuação. Por padrão, o componente reconhecedor de comandos considera as correspondências como válidas somente se sua pontuação for superior a 200, portanto, todas as pontuações que você vir serão de pelo menos 200.

O número após a pontuação é o número de milissegundos desde que o programa foi iniciado. Ao tentar dizer "yes"e "no"percebe-se que a saída se parece com a seguinte:

```
Heard yes (201) @4056ms
Heard no (205) @6448ms
Heard unknown (201) @13696ms
Heard yes (205) @15000ms
Heard yes (205) @16856ms
Heard unknown (204) @18704ms
Heard no (206) @21000ms
```

Na saída apresentada anteriormente, percebe-se que a inferência é realizada em intervalos de tempo e apresenta os casos aonde ela entende que uma palavra foi reconhecida, tal como "yes"ou "no", assim como quando escutou algo que não foi possível interpretar.

Após os experimentos realizados, comprovou-se na prática realizada o correto funcionamento dos sistemas de testes propostos.

No Capitulo 5 discutem-se os resultados do uso do TensorFlow Lite, suas limitações e se faz uma discussão de possiveis aplicações desta tecnologia.

5 ANÁLISES E CONCLUSÕES

Este capitulo prevê uma exploração qualitativa das vantagens e desvantagens do uso do TensorFlow Lite, bem como conclui com algumas sugestões de aplicações e algumas considerações finais.

5.1 Avaliação qualitativa mediante experiência de uso

5.1.1 Adaptabilidade dos exemplos de referência em novas aplicações

Todos os casos de uso para o TensorFlow Lite apresentam um passo a passo no formato do Google Colaboratory que ajudam no processo de uso dos mesmos numa aplicação prática. Cabe destacar que todos eles são desenvolvidos utilizando o treinamento offline, portanto limitados no que tange a auto-descoberta ou o auto-aprendizado (auto-tuning). No entanto, exploram de forma eficiente diversas formas de aquisição de sinais e apresentam uma forma prática de como conseguir gerar uma rede neural eficiente para uma aplicação.

Sendo gerado em código C++, os modelos criados utilizando o TensorFlow Lite são, teoricamente, passíveis de implementação em qualquer microcontrolador que suporta tal linguagem, ou seja, a maioria dos microcontroladores atuais. A biblioteca do TensorFlow já fornece diversas configurações para plataformas diferentes, o que facilita a exploração da ferramenta de forma prática.

Além disso, a biblioteca TensorFlow Lite embarcável para microcontroladores facilita o desenvolvimento de aplicações de IA pois oferece uma interface simples e direta para que programadores usem a mesma e desenvolvam novas aplicações. O modelo gerado de forma offline pode ser carregado na memória do microcontrolador. Então utiliza-se esta biblioteca para conectar as entradas e saídas do processo com o núcleo responsável por fazer a interpretação do modelo e a geração de sinais de forma transparente.

Por fim, entende-se que os casos de uso desenvolvidos servem como exemplos para outras aplicações, especialmente na identificação de voz, ou na identificação de um sinal especifico advindo de sensores.

5.1.2 Geração de novos exemplos de referência

O desenvolvimento de novos casos de uso, utilizando diferentes metodologias ou topologias de redes neurais, carece ainda de pesquisadores capacitados para geração das mesmas, de maneira que possam ser utilizadas de forma prática por desenvolvedores de aplicações. Sendo assim, apesar dos casos de uso existentes abrangerem diversas aplicações possíveis, ainda não é possível cobrir toda e qualquer funcionalidade desejável.

5.1.3 Limitações do TensorFlow Lite

O TensorFlow Lite para Microcontroladores foi projetado considerando as restrições específicas do desenvolvimento de microcontroladores. Caso se esteja trabalhando com dispositivos mais poderosos (por exemplo, um dispositivo Linux incorporado, como o Raspberry Pi), a estrutura padrão do TensorFlow Lite pode ser mais fácil de integrar.

As seguintes limitações devem ser consideradas:

a) Suporte para um subconjunto limitado de operações do TensorFlow;

b) Suporte para um conjunto limitado de dispositivos;

c) API C++ de baixo nível que requer gerenciamento manual de memória;

d) O treinamento no dispositivo não é compatível com as características do dispositivo, por ter baixa capacidade de processamento e memória limitada.

Todos estes casos de uso utilizam o treinamento offline, portanto aplicações que requerem um treinamento online, em tempo real, não são ainda abordadas utilizando esta tecnologia.

5.1.4 Problemas encontrados

Ao longo do desenvolvimento dos casos de uso, alguns problemas foram encontrados, porém a documentação, por ser bem elaborada, facilitou encontrar soluções para cada desafio apresentado. Algumas das complicações encontradas são descritas na sequência.

Apesar da documentação ser recente e bem elaborada, alguns recursos importantes tais como os links do GitHub não são mais facilmente encontrados. Foi necessária uma pesquisa no repositório oficial do GitHub para encontrar todos os casos de uso desenvolvidos ao longo do tempo, assim como os arquivos para embarcar o código nas plataformas de desenvolvimento. Foi realizado uma copia do repositório oficial em aamuller/tensorflow o que ajudou a maner os arquivos de acesso mais simplificado para quem quiser repetir os experimentos.

Alguns softwares de gravação de desenvolvimento são diferentes do que foi documentado por algumas plataformas. Apesar deste não ser um problema relacionado ao TensorFlow em si, torna mais difícil a implementação dos seus casos de uso pois torna a sua documentação obsoleta. O driver do gravador da Spark Fun é o maior exemplo pois houve a necessidade de se utilizar um driver de uma fabricante chinesa, não documentada, para permitir a gravação na plataforma.

5.2 Estudo adicional sugerido

Sugere-se o estudo dos outros casos de uso disponíveis para o TensorFlow Lite pois abrangem exemplos que utilizam diversos outros sensores e componentes muitas vezes utilizadas em aplicações mecatrônicas microcontroladas. Apesar deste trabalho não ter estudado tais aplicações, entende-se que os princípios apresentados nos casos de uso aqui estudados são os mesmos e servem como base para o entendimento de qualquer aplicação.

No entanto, para o desenvolvimento de novos modelos, há a necessidade de um maior entendimento sobre a inteligência artificial em si e suas metodologias, principalmente utilizando-se das tecnologias do TensorFlow para geração de modelos. Este trabalho não seguiu esta linha, sugerindo-se então um estudo mais aprofundado do mesmo que permita a geração de novos modelos, que pode inclusive contribuir com o projeto do TensorFlow de forma direta pois os mesmo busca esta colaboração com pesquisadores e desenvolvedores.

Por fim, sugere-se também o uso destes casos de uso para o desenvolvimento de aplicações microcontroladas de Mecatrônica de borda, que possam usufruir da inteligência artificial para a geração de valor. O tópico a seguir sugere algumas aplicações plausíveis, porém não se estende o suficiente no objetivo de exaurir as possibilidades.

5.2.1 Exemplos para uso do TensorFlow Lite em Projetos Mecatrônicos

Utilizando como base os casos de uso apresentados neste trabalho, assim como os casos de uso da comunidade do TensorFlow Lite, apresentamos algumas aplicações relacionadas a mecatrônica que poderiam utilizar desta tecnologia.

Cabe destacar que o uso do TensorFlow Lite foca nos dispositivos microcontrolados de baixa energia, pouca capacidade de processamento (relativa) e consequentemente menor preço. Portanto, apesar de poder apresentar inúmeras soluções no contexto da mecatrônica em todas as escalas, o foco será naquelas que são determinadas aplicações de borda, pois são as que mais podem se beneficiar desta tecnologia.

Uma aplicação Mecatrônica consiste na integração de elementos de máquina mecânicas com componentes e sistemas eletroeletrônicos sendo gerenciados por um software, inteligente ou não. Neste cenário, entende-se que o software que gerencia o sistema é responsável por orquestrar o funcionamento de todo o sistema, e geralmente é incorporado a este com uma interface muito clara de entradas e saídas. As entradas são geralmente realizadas por meio da aquisição de dados e o uso de sensores. Já as saídas são geralmente comandadas dor drivers de acionadores.

Sendo responsável por gerenciar todo o sistema mecatrônico, o software interage, portanto, diretamente com os resultados provenientes dos sensores deste sistema e comanda os atuadores de forma que os mesmos realizem seu papel da forma mais precisa possível.

Sendo assim, podemos dizer que a inteligência artificial, quando aplicada como um software que interage com um sistema Mecatrônico, deve, tanto interpretar os dados que adquire dos seus sensores, comandos e estados de funcionamento internos, como, de forma orquestrada e inteligente, comandar os seus drivers e atuadores para que interajam com a planta de forma segura, garantindo a funcionalidade esperada.

Para adquirir dados por meio de sensores pode-se utilizar diversos métodos e tipos de sensores diferentes, assim como comandar acionadores também de formas diferentes (proporcional ou liga desliga). Entende-se que uma solução inteligente, no que tange o contexto mecatrônico, além de permitir que o sistema funcione de forma segura e controlada (o que também é possível utilizando métodos convencionais tais como o controle PID), também deve incluir a interpretação das informações de entrada e, de forma inteligente, comande os sinais de saída para os acionadores.

Exemplos de interpretação das informações são observados em todos os casos de uso. Cada um deles aborda formas diferentes de como adquirir os sinais e inferir o que eles representam. O exemplo do reconhecimento de voz permite que se entenda se uma pessoa falou certa palavra, o que poderia gerar um comando, como por exemplo, o acionamento de porta automática. O mesmo caso pode ser levado para diversos outros contextos, à exemplo da detecção de vazamento pneumático ou verificando o ruído gerado pelo mesmo.

Outros exemplos, vinculados a outros sensores tais como o acelerômetro, podem permitir a identificação da frequência de vibração das máquinas ferramenta de uma indústria, apresentando dados úteis para a manutenção destes sistemas.

Diversas aplicações podem ser desenvolvidas seguindo este método de raciocínio: primeiramente entender quais são as capacidades atuais do TensorFlow Lite e finalmente ter a capacidade de desdobrar o mesmo em possíveis soluções na área de Mecatrônica.

Por fim, apesar dos problemas encontrados, como em relação à documentação, mesmo que recente, estar desatualizada, entende-se que o TensorFlow é um projeto estável e a sua ferramenta TensorFlowLite, permite a geração e execução de modelos de inteligência artificial em microcontroladores é uma ferramenta sólida. Sua comunidade, além de extensa, é muito atuante e historicamente fez esta ferramenta ser tão famosa, não sem motivo.

5.3 Sugestões para trabalhos futuros

Sugere-se ainda, o desenvolvimento de uma metodologia que possa facilmente desdobrar os casos de uso existentes para o TensorFlowLite em aplicações para a Mecatrônica, assim como para outras áreas como medicina, entre outras. A metodologia TRIZ poderia ser uma das opções (FRESNER JOHANNES; JANTSCHGI, 2010).

Apesar dos casos de uso atuais serem muito úteis, e relativamente fáceis de serem reproduzidos, o desenvolvimento de aplicações que precisem de uma modelagem diferente ainda requer conhecimento teórico avançado e uma comunidade de pesquisadores capaz de implementar as mesmas. Isso torna um pouco mais difícil o suporte para as funcionalidades onde é desejável um modelo de referência já criado. Ainda assim, os modelos atuais cobrem um bom espectro das aplicações que se referem à Mecatrônica.

Entende-se que quanto mais modelos de referência for desenvolvido, mais a Mecatrônica poderá aproveitar desta tecnologia para o desenvolvimento de aplicações modernas com inteligência artificial embarcada. Esta tecnologia, e principalmente a sua documentação, permitem, de forma prática, resolver problemas até então não resolvidos e/ou inovar nos processos e produtos mecatrônicos embarcando novas capacidades ao sistema.

Ainda assim, seria de grande valia o desenvolvimento de uma metodologia que facilitasse o desdobramento dos casos de uso do TensorFlowLite em possíveis soluções de funcionalidades relacionadas a mecatrônica ou mesmo outras áreas do conhecimento tais como a medicina. Isso facilitaria os desenvolvedores de produto na operação pelo TensorFlow Lite como uma ferramenta de planejamento de produtos.

REFERÊNCIAS

AI WIKI. ***Weights and Biases***. 2022. Disponível em: https://machine-learning.paperspace.com/wiki/weights-and-biases. Acesso em: 5 dec 2022. 37

ARAÚJO ANEIDE OLIVEIRA; OLIVEIRA, M. C. ***Tipos de pesquisa***. [S.l.]: Departamento de Controladoria e Contabilidade da USP, 1997. 27

ARM. ***ARM Processors***. 2022. Disponível em: https://developer.arm.com/ip-products/processors/cortex-m. Acesso em: 5 dec 2022. 20

ESPRESSIF. ***ESP32 Wi-Fi Bluetooth MCU I Espressif Systems***. 2022. Disponível em: https://www.espressif.com/en/products/hardware/esp32/overview. Acesso em: 5 dec 2022. 20

FRESNER JOHANNES; JANTSCHGI, J. B. S. B. J. K. C. ***The theory of inventive problem solving (TRIZ) as option generation tool within cleaner production projects***. 2010. 76

GITHUB. ***New Platform Support : TensorFlow Lite***. 2021. Disponível em: https://github.com/tensorflow/tflite-micro/blob/main/tensorflow/lite/micro/docs/new_platform_support.md. Acesso em: 5 dec 2022. 25

GOOGLE. ***Google Colab***. 2022. Disponível em: https://colab.research.google.com/. Acesso em: 5 dec 2022. 28

GOOGLE. ***TensorFlow Lite for Microcontrollers - Experiments with Google***. 2022. Disponível em: https://experiments.withgoogle.com/collection/tfliteformicrocontrollers. Acesso em: 5 dec 2022. 25

MEDIUM. ***Running TensorFlow Lite for Microcontrollers on Contiki-NG***. 2021. Disponível em: https://atiselsts.medium.com/running-tensorflow-lite-for-microcontrollers-on-contiki-ng-d938f25193f5. Acesso em: 5 dec 2022. 19, 21

TENSORFLOW. ***API Documentation : TensorFlow v2.11.0***. 2022. Disponível em: https://www.tensorflow.org/api_docs. Acesso em: 5 dec 2022. 19

TENSORFLOW. ***Build and convert models : TensorFlow Lite***. 2022. Disponível em: https://www.tensorflow.org/lite/microcontrollers/build_convert#model_conversion. Acesso em: 5 dec 2022. 20

TENSORFLOW. ***Convert to a C Array : TensorFlow Lite***. 2022. Disponível em: https://www.tensorflow.org/lite/microcontrollers/build_convert#convert_to_a_c_array. Acesso em: 5 dec 2022. 21

TENSORFLOW. ***Launching the Speech Commands Dataset***. 2022. Disponível em: https://ai.googleblog.com/2017/08/launching-speech-commands-dataset.html. Acesso em: 5 dec 2022. 66

TENSORFLOW. ***Operation Support : TensorFlow Lite***. 2022. Disponível em: https://www.tensorflow.org/lite/microcontrollers/build_convert#operation_support. Acesso em: 5 dec 2022. 20

TENSORFLOW. ***TensorFlow***. 2022. Disponível em: https://www.tensorflow.org/. Acesso em: 5 dec 2022. 19

TENSORFLOW. ***TensorFlow Lite for Microcontrollers***. 2022. Disponível em: https://www.tensorflow.org/lite/microcontrollers. Acesso em: 5 dec 2022. 19

TENSORFLOW. ***Train a Simple TensorFlow Lite for Microcontrollers model***. 2022. Disponível em: https://github.com/tensorflow/tflite-micro/blob/main/tensorflow/lite/micro/examples/hello_world/train/train_hello_world_model.ipynb. Acesso em: 5 dec 2022. 80, 88

TINYMLEDU. ***The Tiny Machine Learning Open Education Initiative (TinyMLedu)***. 2022. Disponível em: http://tinyml.seas.harvard.edu/. Acesso em: 5 dec 2022. 26

WARDEN, P. S. D. ***TinyML***. [S.l.]: O'Reilly Media, 2019. 28, 30, 31, 32, 38, 48, 49, 67, 69, 70, 92, 95, 97, 98, 99, 100, 101, 102, 103, 105, 108

ANEXOS

ANEXO A – MODELO PARA GERAÇÃO DA FUNÇÃO SENO

Este código foi adaptado de (TENSORFLOW, 2022h) onde foi traduzido e ampliado para apresentar algumas informações adicionais e resultados mais adequados.

```
# Definir caminhos para arquivos de modelo
import os
MODELS_DIR = 'models/'
if not os.path.exists(MODELS_DIR):
    os.mkdir(MODELS_DIR)
MODEL_TF = MODELS_DIR + 'model'
MODEL_NO_QUANT_TFLITE = MODELS_DIR + 'model_no_quant.tflite'
MODEL_TFLITE = MODELS_DIR + 'model.tflite'
MODEL_TFLITE_MICRO = MODELS_DIR + 'model.cc'

# Instalar dependências
! pip install tensorflow==2.4.0

# Importar dependências

# TensorFlow é uma biblioteca de aprendizado de máquina de
# código aberto
import tensorflow as tf
# Keras é a API de alto nível do TensorFlow para aprendizado profundo
from tensorflow import keras
# Numpy é uma biblioteca de matemática
import numpy as np
# Pandas é uma biblioteca de manipulação de dados
import pandas as pd
# Matplotlib é uma biblioteca de gráficos
import matplotlib.pyplot as plt
# Math é a biblioteca de matemática do Python
import math

# Gerar dados: o conjunto de dados ira possui a quantidade de SAMPLES

SAMPLES = 1000

# Define um valor SEED para que possamos ter o mesmo resultado de número
# aleatórios se o código for executado novamente. O valor de semente é
# um valor base usado por um gerador pseudo-aleatório para produzir
# números aleatórios.

SEED = 1200
np.random.seed(SEED)
tf.random.set_seed(SEED)
```

```
# Gere um conjunto uniformemente distribuído de números aleatórios
# no intervalo de 0 to 2, com quantidade de pontos definidas como SAMPLES.
x_values = np.random.uniform(low=0, high=2*math.pi, size=SAMPLES)

# Embaralhe os valores para garantir que não estão em ordem.
np.random.shuffle(x_values)

# Calcule os valores de seno correspondentes
y_values = np.sin(x_values)

# Plote os resultados

plt.rcParams['figure.figsize'] = [15, 10]
plt.plot(x_values, y_values, 'r.')
plt.show()

# Adicionar Ruído aleatório a cada valor de y
y_values += 0.1 * np.random.randn(*y_values.shape)
# Plotar nossos dados
plt.rcParams['figure.figsize'] = [15, 10]
plt.plot(x_values, y_values, 'r.')
plt.show()

# Dividir os dados
# Usaremos 60% de nossos dados para treinamento e 20% para teste.
# Os 20% restantes será usado para validação.

TRAIN_SPLIT =

int(0.6 * SAMPLES)

TEST_SPLIT = int(0.2 * SAMPLES + TRAIN_SPLIT)

# Use np.split para dividir nossos dados em três partes.
# O segundo argumento para np.split é um array de índices onde os dados
# serão divididos. Como existem dois índeces, os dados serão divididos
# em três pedaços.

x_train, x_validate, x_test = np.split(x_values, [TRAIN_SPLIT, TEST_SPLIT])
y_train, y_validate, y_test = np.split(y_values, [TRAIN_SPLIT, TEST_SPLIT])

# Verifique se nossas divisões somam corretamente

assert (x_train.size + x_validate.size + x_test.size) == SAMPLES

# Plotar os dados
```

```
plt.rcParams['figure.figsize'] = [15, 10]
plt.plot(x_train, y_train, 'r.', label="Treinamento")
plt.plot(x_test, y_test, 'b.', label="Teste")
plt.plot(x_validate, y_validate, 'g.', label="Validação")
plt.legend()
plt.show()

# Construir o modelo

# Usaremos o Keras para criar uma arquitetura de modelo simples

from tf.keras import layers

model_1 = tf.keras.Sequential()

# A primeira camada recebe uma entrada escalar e a alimenta através de
# 16 "neurônios". Os neurônios decidem se ativam com base na função de
# ativação 'relu'.

model_1.add(layers.Dense(16, activation='relu', input_shape=(1,)))

# A camada final é um único neurônio, já que queremos produzir
# um único valor.

model_1.add(layers.Dense(1))

# Compile o modelo usando um otimizador padrão e uma função de perda

# para regressão.

model_1.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])

# Imprima a arquitetura do modelo.

tf.keras.utils.plot_model(model_1, show_shapes=True, show_layer_names=True)

# Treinar o modelo

# Treine o modelo em nossos dados de treinamento enquanto
# valida em nosso conjunto de validação
history_1 = model_1.fit(x_train, y_train, epochs=500, batch_size=64,
                        validation_data=(x_validate, y_validate))

# Resultados

# Desenhe um gráfico da perda, que é a distância entre
# os valores previstos e reais durante o treinamento e validação.
train_loss = history_1.history['loss']
```

```
val_loss = history_1.history['val_loss']

epochs = range(1, len(train_loss) + 1)

plt.plot(epochs, train_loss, 'b.', label='Perda no treinamento')
plt.plot(epochs, val_loss, 'g', label='Perda na validação')
plt.xlabel('Épocas')
plt.ylabel('Perda')
plt.legend()
plt.show()

# Exclua as primeiras épocas para que o gráfico seja mais fácil de ler
SKIP = 50

plt.plot(epochs[SKIP:], train_loss[SKIP:], 'b.', label='Perda no treinamento')
plt.plot(epochs[SKIP:], val_loss[SKIP:], 'g.', label='Perda na validação')
plt.xlabel('Épocas')
plt.ylabel('Perda')
plt.legend()
plt.show()

# Desenhe um gráfico de erro absoluto médio, que é outra maneira de
# medindo a quantidade de erro na previsão.
train_mae = history_1.history['mae']
val_mae = history_1.history['val_mae']

plt.plot(epochs[SKIP:], train_mae[SKIP:], 'g.', label='MAE Treinamento')
plt.plot(epochs[SKIP:], val_mae[SKIP:], 'b.', label='MAE Validação')
plt.xlabel('Epocas')
plt.ylabel('MAE')
plt.legend()
plt.show()

# Calcule e imprima a perda em nosso conjunto de dados de teste
test_loss, test_mae = model_1.evaluate(x_test, y_test)

# Faça previsões com base em nosso conjunto de dados de teste
y_test_pred = model_1.predict(x_test)

# Representar graficamente as previsões em relação aos valores reais
plt.rcParams['figure.figsize'] = [15, 10]
plt.clf()
plt.plot(x_test, y_test, 'r.', label='Valores atuais')
plt.plot(x_test, y_test_pred, 'b.', label='Previsões')
plt.legend()
plt.show()

# Criando um segundo modelo mais elaborado
```

```
model = tf.keras.Sequential()

# A primeira camada pega uma entrada escalar e a alimenta através
# de 16 "neurônios". Os neurônios decidem se devem ser ativados
# com base na função de ativação 'relu'.
model.add(keras.layers.Dense(16, activation='relu', input_shape=(1,)))

# A nova segunda e terceira camada ajudarão a rede a aprender
# representações mais complexas
model.add(keras.layers.Dense(16, activation='relu'))

# A camada final é um único neurônio, uma vez que queremos produzir
# um único valor
model.add(keras.layers.Dense(1))

# Compile o modelo usando o otimizador padrão 'adam' e o erro quadrado
# médio ou a função de perda 'mse' para regressão.
model.compile(optimizer='adam', loss="mse", metrics=["mae"])
tf.keras.utils.plot_model(model_1, show_shapes=True, show_layer_names=True)

# Treine o modelo
history = model.fit(x_train, y_train, epochs=500, batch_size=64,
                    validation_data=(x_validate, y_validate))

# Treine o modelo
model.save(MODEL_TF)

# Resultados

# Desenhe um gráfico da perda, que é a distância entre
train_loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(train_loss) + 1)

# Exclua as primeiras épocas para que o gráfico seja mais fácil de ler
SKIP = 100

plt.rcParams['figure.figsize'] = [15, 10]
plt.subplot(1, 2, 1)

plt.plot(epochs[SKIP:], train_loss[SKIP:], 'g.', label='Perda no treinamento')
plt.plot(epochs[SKIP:], val_loss[SKIP:], 'b.', label='Perda na validação')
plt.xlabel('Epocas')
plt.ylabel('Perdas')
plt.legend()
```

```
plt.subplot(1, 2, 2)

# Desenhe um gráfico de erro absoluto médio, que é outra maneira de
# medindo a quantidade de erro na previsão.
train_mae = history.history['mae']
val_mae = history.history['val_mae']

plt.plot(epochs[SKIP:], train_mae[SKIP:], 'g.', label='MAE Treinamento')
plt.plot(epochs[SKIP:], val_mae[SKIP:], 'b.', label='MAE Validação')
plt.xlabel(‘Epocas’)
plt.ylabel('MAE')
plt.legend()

plt.tight_layout()

# Calcule e imprima a perda em nosso conjunto de dados de teste
test_loss, test_mae = model.evaluate(x_test, y_test)

# Faça previsões com base em nosso conjunto de dados de teste
y_test_pred = model.predict(x_test)

# Representar graficamente as previsões em relação aos valores reais
plt.clf()
plt.plot(x_test, y_test, 'b.', label=‘Valores atuais’)
plt.plot(x_test, y_test_pred, 'r.', label=‘Previsões’)
plt.legend()
plt.show()

# Converter o modelo para o formato TensorFlow Lite sem quantização
converter = tf.lite.TFLiteConverter.from_saved_model(MODEL_TF)
model_no_quant_tflite = converter.convert()

# Salve o modelo em disco
open(MODEL_NO_QUANT_TFLITE, "wb").write(model_no_quant_tflite)

# Converter o modelo para o formato TensorFlow Lite com quantização
def representative_dataset():
  for i in range(500):
    yield([x_train[i].reshape(1, 1)])
# Defina o sinalizador de otimização.
converter.optimizations = [tf.lite.Optimize.DEFAULT]
# Impor quantização somente de inteiro
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
converter.inference_input_type = tf.int8
converter.inference_output_type = tf.int8
# Forneça um conjunto de dados representativo para garantir
# que quantifiquemos corretamente.
converter.representative_dataset = representative_dataset
```

```
model_tflite = converter.convert()

# Salve o modelo em disco
open(MODEL_TFLITE, "wb").write(model_tflite)

# Definimos as funções de previsão (para previsões) e
# avaliamos (para perda) para modelos TFLite.

def predict_tflite(tflite_model, x_test):
  # Prepare the test data
  x_test_ = x_test.copy()
  x_test_ = x_test_.reshape((x_test.size, 1))
  x_test_ = x_test_.astype(np.float32)

  # Initialize the TFLite interpreter
  interpreter = tf.lite.Interpreter(model_content=tflite_model,
    experimental_op_resolver_type=tf.lite.experimental.OpResolverType.BUILTIN_REF)
  interpreter.allocate_tensors()

  input_details = interpreter.get_input_details()[0]
  output_details = interpreter.get_output_details()[0]

  # Se necessário, quantize a camada de entrada (de float para inteiro)
  input_scale, input_zero_point = input_details["quantization"]
  if (input_scale, input_zero_point) != (0.0, 0):
    x_test_ = x_test_ / input_scale + input_zero_point
    x_test_ = x_test_.astype(input_details["dtype"])

  # Invoque o intérprete
  y_pred = np.empty(x_test_.size, dtype=output_details["dtype"])
  for i in range(len(x_test_)):
    interpreter.set_tensor(input_details["index"], [x_test_[i]])
    interpreter.invoke()
    y_pred[i] = interpreter.get_tensor(output_details["index"])[0]

  # Se necessário, desquantizou a camada de saída (do inteiro ao flutuador)
  output_scale, output_zero_point = output_details["quantization"]
  if (output_scale, output_zero_point) != (0.0, 0):
    y_pred = y_pred.astype(np.float32)
    y_pred = (y_pred - output_zero_point) * output_scale

  return y_pred

def evaluate_tflite(tflite_model, x_test, y_true):
  global model
  y_pred = predict_tflite(tflite_model, x_test)
  loss_function = tf.keras.losses.get(model.loss)
  loss = loss_function(y_true, y_pred).numpy()
```

```
  return loss

# Calcular previsões
y_test_pred_tf = model.predict(x_test)
y_test_pred_no_quant_tflite = predict_tflite(model_no_quant_tflite, x_test)
y_test_pred_tflite = predict_tflite(model_tflite, x_test)

# Calcule a perda
loss_tf, _ = model.evaluate(x_test, y_test, verbose=0)
loss_no_quant_tflite = evaluate_tflite(model_no_quant_tflite, x_test, y_test)
loss_tflite = evaluate_tflite(model_tflite, x_test, y_test)

# Calcule a perda
df = pd.DataFrame.from_records(
    [["TensorFlow", loss_tf],
     ["TensorFlow Lite", loss_no_quant_tflite],
     ["TensorFlow Lite Quantized", loss_tflite]],
     columns = ["Model", "Loss/MSE"], index="Model").round(4)
df

# Calcular tamanho
size_tf = os.path.getsize(MODEL_TF)
size_no_quant_tflite = os.path.getsize(MODEL_NO_QUANT_TFLITE)
size_tflite = os.path.getsize(MODEL_TFLITE)

# Compare o tamanho
pd.DataFrame.from_records(
    [["TensorFlow", f"{size_tf} bytes", ""],
     ["TensorFlow Lite", f"{size_no_quant_tflite} bytes ",
       f"(reduced by {size_tf - size_no_quant_tflite} bytes)"],
     ["TensorFlow Lite Quantized", f"{size_tflite} bytes",
       f"(reduced by {size_no_quant_tflite - size_tflite} bytes)"]],
     columns = ["Model", "Size", ""], index="Model")

# Gerar um TensorFlow Lite para Microcontroladores Modelo

# Instale xxd se não estiver disponível
!apt-get update && apt-get -qq install xxd
# Converter para um arquivo de origem C, ou seja, um TensorFlow Lite
# para Microcontroladores modelo
!xxd -i {MODEL_TFLITE} > {MODEL_TFLITE_MICRO}
# Atualizar nomes de variáveis
REPLACE_TEXT = MODEL_TFLITE.replace('/', '_').replace('.', '_')
!sed -i 's/'{REPLACE_TEXT}'/g_model/g' {MODEL_TFLITE_MICRO}

# Imprimir o arquivo de origem C
! cat {MODEL_TFLITE_MICRO}
```

ANEXO B – MODELO WAKE WORD

Este código foi adaptado de (TENSORFLOW, 2022h) onde foi traduzido e ampliado para apresentar algumas informações adicionais e resultados mais adequados.

```
import os

# As seguintes linhas os.environ podem ser personalizadas...
# Uma lista delimitada por vírgulas das palavras para as quais
você deseja treinar.  As opções são: sim, não, para cima, para baixo,
# esquerda, direita, on, off, stop, go. Todas as outras palavras serão
# usadas para treinar uma categoria "unknown".
os.environ["WANTED_WORDS"] = "yes,no"

# O número de etapas e as taxas de aprendizado podem ser especificados
# como separados por vírgulas
# listas para definir a taxa em cada etapa. Por exemplo
# TRAINING_STEPS=15000,3000 e LEARNING_RATE=0.001,0.0001
# irão executar 18,000 loops de treinamento no total
# com uma taxa de 0,001 para o primeiro
# 15.000 e 0,0001 para os 3.000 finais.
os.environ["TRAINING_STEPS"]="15000,3000"
os.environ["LEARNING_RATE"]="0.001,0.0001"

# Calcule o número total de etapas, que é usado para identificar o ponto
# de verificação do nome do arquivo.
total_steps = sum(map(lambda string: int(string),
                  os.environ["TRAINING_STEPS"].split(",")))
os.environ["TOTAL_STEPS"] = str(total_steps)

# Imprima a configuração para confirmá-la
!echo "Training these words: ${WANTED_WORDS}"
!echo "Training steps in each stage: ${TRAINING_STEPS}"
!echo "Learning rate in each stage: ${LEARNING_RATE}"
!echo "Total number of training steps: ${TOTAL_STEPS}"

# Em seguida, vamos instalar uma compilação de GPU do TensorFlow, para
# que possamos usar a aceleração de GPU  para treinamento. Também
# clonamos o repositório TensorFlow, que contém os scripts que treinam
# e congelam o modelo.
! pip install -q tf-nightly-gpu==1.15.0.dev20190729
! git clone https://github.com/tensorflow/tensorflow

# Exclua todos os logs antigos de execuções anteriores e carregue
# o TensorBoard
!rm -rf /content/retrain_logs
```

```
# Carregar TensorBoard
%load_ext tensorboard
%tensorboard --logdir /content/retrain_logs

# Comece o treinamento
!python tensorflow/tensorflow/examples/speech_commands/train.py \
--model_architecture=tiny_conv --window_stride=20 --preprocess=micro \
--wanted_words=${WANTED_WORDS} --silence_percentage=25 --unknown_percentage=25 \
--quantize=1 --verbosity=WARN --how_many_training_steps=${TRAINING_STEPS} \
--learning_rate=${LEARNING_RATE} --summaries_dir=/content/retrain_logs \
--data_dir=/content/speech_dataset --train_dir=/content/speech_commands_train \

# Quando o treinamento estiver concluído, execute a seguinte célula
# para congelar o gráfico
!python tensorflow/tensorflow/examples/speech_commands/freeze.py \
--model_architecture=tiny_conv --window_stride=20 --preprocess=micro \
--wanted_words=${WANTED_WORDS} --quantize=1 --output_file=/content/tiny_conv.pb \
--start_checkpoint=/content/speech_commands_train/tiny_conv.ckpt-${TOTAL_STEPS}

# Converter o modelo
# Execute esta célula para usar o conversor TensorFlow Lite para converter
# o gráfico congelado no formato TensorFlow Lite, totalmente quantizado
# para uso com dispositivos embarcados.
!toco \
--graph_def_file=/content/tiny_conv.pb --output_file=/content/tiny_conv.tflite \
--input_shapes=1,1960 --input_arrays=Reshape_1 --output_arrays='labels_softmax' \
--inference_type=QUANTIZED_UINT8 --mean_values=0 --std_dev_values=9.8077

# A célula a seguir imprimirá o tamanho do modelo, que será inferior a 20
# kilobytes.
import os
model_size = os.path.getsize("/content/tiny_conv.tflite")
print("Model is %d bytes" % model_size)

# Finalmente, usamos xxd para transformar o modelo em um arquivo de origem que
# pode ser incluído em um projeto C++ e carregado pelo TensorFlow Lite para
# Microcontroladores.
# Instale xxd se não estiver disponível
! apt-get -qq install xxd
# Salve o arquivo como um arquivo de origem C
! xxd -i /content/tiny_conv.tflite > /content/tiny_conv.cc
# Imprimir o arquivo de origem
! cat /content/tiny_conv.cc
```

ANEXO C – REVISÃO INTEGRATIVA

O objetivo deste anexo é apresentar a Inteligência Artificial, especificamente o Aprendizado de Máquina, de forma simplificada, com foco prático, e mostrar como esses conceitos podem ser implementados com o framework TensorFlow Lite em microcontroladores.

C.1 Dispositivos Embarcados e a Mecatrônica

Microcontroladores são usados em mecatrônica principalmente para aplicações que requerem baixo consumo de energia, normalmente menos de um miliwatt. Sendo assim, precisamos de ferramentas para especificar e implementar produtos de baixo consumo para dispositivos embarcados. Mais recentemente, há o interesse de executar modelos de inteligência artificial nestes dispositivos aonde se demonstra importância do TensorFlow Lite, que além de permitir a execução destes modelos em microcontroladores, ainda se adequa aos requisitos de energia de diversas aplicações.

Há apenas alguns anos atrás, estes dispositivos constituíam-se de máquinas de 8 bits e usavam cadeias de ferramentas proprietárias. Parecia muito intimidador começar a trabalhar com qualquer um deles. Um grande passo à frente se deu com a introdução do Arduino por possuir uma IDE amigável ao usuário em conjunto com um hardware padronizado. Desde então, as CPU's de 32 bits se tornaram o padrão, em grande parte graças à série de chips Cortex-M da ARM. Isto possibilitou uma grande simplificação no processo de desenvolvimento.

No entanto, os dispositivos embarcados ainda apresentam algumas restrições. Frequentemente, possuem apenas algumas centenas de kilobytes de RAM, ou às vezes muito menos do que isso, e têm quantidades semelhantes de memória Flash (não volátil) para programas e armazenamento de dados. Uma velocidade de *clock* de apenas dezenas de mega-hertz também é comum nestes dispositivos.

C.1.1 Desenvolvimento Embarcado e o Aprendizado de Máquina

Antes, é preciso esclarecer que o Aprendizado de Máquina é um subconjunto da Inteligência Artificial, que é um termo amplo e que engloba várias técnicas. Atualmente é possível executar software estruturado com Aprendizado de Máquina em microcontroladores, apesar de ser um campo ainda pouco explorado. A estrutura do software TensorFlow Lite que usamos no decorrer deste trabalho tem uma *API* estável e uma grande comunidade de discussão e suporte. Há muito material disponível na internet e há uma tendência desta disponibilidade se intensificar nos próximos anos.

A primeira e principal razão para a escolha de microcontroladores é sua popularidade em várias áreas de aplicação, como o setor automotivo, eletrônicos de

consumo, utensílios de cozinha, saúde e telecomunicações. Os microcontroladores estão por toda parte e, muitas vezes, invisíveis em dispositivos eletrônicos usados no dia-a-dia.

Com a popularização do IoT, os microcontroladores tiveram um crescimento exponencial de mercado. Em 2018, a empresa de pesquisa de mercado IDC relatou 28,1 bilhões de microcontroladores vendidos em todo o mundo e previu um crescimento de 38,2 bilhões até 2023. São números impressionantes, considerando que os mercados de smartphones e computadores registraram 1,5 bilhão e 67,2 milhões de dispositivos, respectivamente, vendidos no mesmo ano.

Desta forma, o TensorFlow Lite representa um avanço significativo para dispositivos IoT, permitindo impulsionar a proliferação de pequenos controladores conectados, capazes de executar tarefas de aprendizado de máquina localmente.

A segunda razão para escolher microcontroladores é que eles são baratos, fáceis de programar e poderosos o suficiente para executar algoritmos sofisticados de aprendizado de máquina. No entanto, há que se notar que a computação em nuvem é uma realidade cada vez mais presente e pode ser viável executar o aprendizado de máquina em nuvem ao invés de executar localmente nos microcontroladores.

Os microcontroladores são tipicamente dispositivos de computação pequenos e de baixa potência, são incorporados ao hardware e requerem computação básica para o desenvolvimento de aplicações. Ao trazer o aprendizado de máquina para os microcontroladores, pode-se aumentar a inteligência de bilhões de dispositivos que potencialmente existirão mundo afora, incluindo eletrodomésticos e dispositivos de IoT, sem depender de hardware caro ou conexões de internet confiáveis, sujeitas a largura de banda, alta latência e restrições de energia. Isso também pode ajudar a preservar a privacidade, já que nenhum dado sai do dispositivo.

Já é possível imaginar aparelhos inteligentes que podem se adaptar à sua rotina diária, sensores industriais inteligentes que entendem a diferença entre problemas de operação e brinquedos que podem ajudar as crianças a aprender de maneira divertida e prazerosa, tudo isso acoplados ao próprio dispositivo sensor ou aos brinquedos, sem a necessidade de um pós-processamento para obter esses parâmetros. Esse processamento no próprio microcontrolador, na ponta do sistema de aquisição e controle, é conhecido como *Border Computing* ou simplesmente Processamento de Borda.

C.1.2 Porquê executar Aprendizado de Máquina na borda

Há três respostas principais para essa pergunta: latência, consumo de energia e privacidade:

a) Redução da latência: o envio e recebimento de dados de e para a nuvem

não é instantâneo e pode afetar os aplicativos que devem responder de forma confiável dentro de um determinado período de tempo.

b) Redução do consumo de energia: enviar e receber dados de e para a nuvem não é eficiente no consumo de energia, mesmo ao usar protocolos de comunicação de baixo consumo, como o “Bluetooth-LE”.

c) Privacidade: é evidente a redução da privacidade ao se enviar e receber dados de e para a nuvem.

C.2 O que é o Aprendizado de Máquina?

Existem poucas áreas em tecnologia com a mística que envolve o Aprendizado de Máquina. Mesmo sendo um engenheiro experiente em outra área, o Aprendizado de Máquina pode parecer um assunto denso, com muitos requisitos de conhecimento para a sua compreensão. Muitos desenvolvedores se sentem desencorajados quando começam a ler sobre o tema e encontram explicações que invocam artigos acadêmicos, bibliotecas obscuras codificadas em Python e matemática avançada. Pode parecer assustador até mesmo saber por onde começar.

> Na realidade, o aprendizado de máquina pode ser simples de entender e acessível a qualquer pessoa com um editor de texto. Depois de aprender algumas ideias-chave, você pode usá-lo facilmente em seus próprios projetos. Por trás de toda a mística está um conjunto útil de ferramentas para resolver vários tipos de problemas. Às vezes pode parecer mágica, mas é apenas um código e você não precisa ser um PhD para trabalhar com isso (WARDEN, 2019, pg. 12).

C.2.1 Como Identificar um Caso de Uso para Aprendizado de Máquina?

Imagine que você possui uma máquina que fabrica dispositivos eletrônicos portáteis. Às vezes, ela apresenta defeito e é caro de consertar. Talvez, se forem coletados dados sobre a máquina durante a operação, seja possível prever quando ela está prestes a apresentar defeito e interromper a operação antes que ocorram danos. Por exemplo, pode-se registrar sua taxa de produção, sua temperatura e o quanto está vibrando. Pode ser que alguma combinação desses fatores indique um problema iminente. Mas como seria possível descobrir isso?

Este é um exemplo do tipo de problema que o Aprendizado de Máquina foi projetado para resolver. Fundamentalmente, é uma técnica que tem se mostrado muito eficaz para prever eventos com base em observações anteriores. Coletamos dados sobre o desempenho de nossas máquinas na fábrica e, em seguida, criamos um programa de computador que analisa esses dados e os utiliza para prever estados futuros.

Criar um programa de Aprendizado de Máquina é diferente do processo usual de escrever código. Em um software tradicional, um programador projeta um algoritmo que recebe uma entrada, aplica várias regras e retorna uma saída. As operações internas do algoritmo são planejadas pelo programador e implementadas explicitamente por meio de linhas de código. Para prever avarias em uma máquina fabril, por exemplo, o programador precisaria entender quais medições nos dados indicam um problema e escrever um código que as verificasse deliberadamente.

Essa abordagem funciona bem para muitos problemas. Por exemplo, sabe-se que a água ferve a 100℃ ao nível do mar, então é fácil escrever um programa que possa prever se a água está fervendo com base em sua temperatura e altitude atuais. Mas, em muitos casos, pode ser difícil saber a combinação exata de fatores que prevê um determinado estado. Para continuar com este exemplo de máquina fabril, pode haver várias combinações diferentes de taxa de produção, temperatura e nível de vibração que podem indicar um problema, mas não são imediatamente óbvias ao se observar os dados.

Para criar um programa voltado à aprendizado de máquina, um programador deve inserir dados em um tipo especial de algoritmo e permitir que o algoritmo descubra as próprias regras, as quais irão reger o seu funcionamento. Isso representa a possibilidade de criar programas que fazem previsões com base em dados complexos sem ter que entender toda a complexidade da relação dos dados em si. O algoritmo de Aprendizado de Máquina constrói um modelo do sistema com base nos dados que fornecemos, por meio de um processo denominado como treinamento. O modelo, desta forma, é um tipo de programa de computador, e em posse do mesmo, chamamos de inferência o processo de inserir os dados no modelo para fazer as previsões.

Há muitas abordagens diferentes para Aprendizado de Máquina. Uma das mais usadas é a *Deep Learning*[1], baseada em uma ideia simplificada de como o cérebro humano pode funcionar. Nesse modo, uma rede de neurônios virtuais (simulados e representados por matrizes de números) é treinada para modelar as relações entre várias entradas e saídas.

Também deve-se entender que diferentes arquiteturas, ou arranjos de neurônios virtuais, são úteis para diferentes tarefas. Por exemplo, algumas arquiteturas são excelentes para extrair significado de dados de imagem, enquanto outras arquiteturas funcionam melhor para prever o próximo valor em uma sequência numérica.

[1] *Deep Learning* ou *Aprendizado Profundo, é uma técnica de aprendizado de máquina que ensina os computadores a fazer o que é natural para os humanos: aprender pelo exemplo.*

C.2.2 O Processo de Desenvolvimento com Aprendizado Profundo

Na seção anterior, descreveu-se um possível cenário no qual o Aprendizado Profundo tem uma promissora possibilidade de aplicação, ou seja, prever quando é provável que uma máquina fabril venha a apresentar defeito. Para que isso seja implementado é necessário um mínimo conjunto de tarefas organizado, que facilitará a abordagem:

a) Definir o objetivo

b) Definir o conjunto de dados

c) Selecionar, Coletar e Classificar os dados

d) Projetar a arquitetura do modelo

e) Treinar o modelo

f) Converter o modelo

g) Realizar a inferência

h) Testar, solucionar problemas e validar

A seguir são detalhadas cada uma destas tarefas.

C.2.2.1 Definir o objetivo

Ao projetar qualquer tipo de algoritmo, é fundamental começar estabelecendo exatamente o que se deseja que ele faça. Isto também se aplica aos algoritmos de aprendizado de máquina. É imprescindível definir bem o que se deseja prever para poder decidir quais dados coletar e qual arquitetura de modelo usar.

No exemplo em questão, deseja-se prever se a máquina fabril está prestes a apresentar defeito. A forma tradicional de abordar este problema é expressá-lo como um problema de classificação. A classificação é uma tarefa de aprendizado de máquina que a partir de um conjunto de dados de entrada, retorna a probabilidade de que esses dados se encaixem em uma determinada classe, considerando uma entre várias classes conhecidas. Para o exemplo da máquina fabril, pode-se ter duas classes: “normal”, significando que está operando sem problemas, e “anormal”, significando estar dando sinais de que pode vir a falhar em breve. Desta forma, fica definido o objetivo: criar um modelo que classifique os dados de entrada como “normais” ou “anormais”.

C.2.2.2 Definir o conjunto de dados

É muito provável que haja muitos dados disponíveis a respeito da unidade fabril onde a máquina está localizada, desde a temperatura de operação da própria máquina, até o tipo de comida que foi servida no refeitório em um determinado dia,

algo aparentemente desconexo mas que pode ser levado em consideração. Dado o objetivo que acabamos de estabelecer, podemos começar a identificar quais são os dados relevantes para o problema.

C.2.2.3 Selecionar os dados

Os modelos de aprendizado profundo podem ser construídos também para aprender a ignorar dados ruidosos ou irrelevantes. Diante disso, pode ser mais eficaz treinar o modelo apenas usando informações relevantes para resolver o problema. Como é improvável que a comida do refeitório, por exemplo, tenha impacto relevante no funcionamento da máquina, provavelmente podemos excluí-la do nosso conjunto de dados. Caso não tomemos esta precaução, o modelo precisará aprender a negar essa entrada irrelevante e pode ficar vulnerável a aprender associações espúrias. Uma "coincidência" pode levar, por exemplo, a máquina sempre quebrar nos dias em que uma macarronada é servida no refeitório.

É muito importante tentar combinar a experiência das pessoas que tenham experiência no assunto em questão, para decidir se um determinado dado deve ser incluído ou não. Também é possível usar métodos estatísticos para identificar quais dados são mais significativos. Se não houver certeza sobre a inclusão de uma determinada fonte de dados, uma abordagem eficiente é treinar dois ou mais modelos e verificar qual funciona melhor.

Vamos supor que foram identificados como os dados mais promissores a taxa de produção, a temperatura e a vibração. O próximo passo é coletar alguns dados para que se possa treinar um modelo.

> É muito importante que os dados escolhidos também estejam disponíveis quando as previsões forem feitas. Por exemplo, ao se decidir treinar o modelo com leituras de temperatura, precisa-se fornecer as leituras de temperatura exatamente dos mesmos locais físicos para quando for executada a inferência. Deve ser assim porque o modelo aprende a entender como suas entradas podem prever suas saídas. Se originalmente treina-se o modelo com dados de temperatura do interior da máquina, é improvável que a execução do modelo funcione adequadamente, com dados da temperatura ambiente. (WARDEN, 2019, p. 20).

C.2.2.4 Coletar os dados

Não é uma tarefa fácil saber com precisão quantos dados são necessários para treinar um modelo de forma eficaz, pois há dependência de muitos fatores, tais como: a complexidade das relações entre as variáveis, a quantidade de ruído contida nos dados e a facilidade com que as classes podem ser distinguidas. Entretanto, no geral, quanto mais dados, melhor o resultado final.

Os dados coletados devem representar toda a gama de condições e eventos que possam ocorrer no sistema em estudo. No caso do exemplo da máquina, esta pode apresentar defeito de várias maneiras diferentes. É importante certificar-se de capturar os dados relacionados a cada tipo de falha. Assim, se uma variável mudar de forma natural ao longo do tempo envolvido na análise, é importante coletar dados que representem o intervalo completo. Por exemplo, no caso da máquina, se a temperatura aumentar em dias quentes, devem ser incluídos os dados de temperatura no inverno e no verão. Essa diversidade ajudará o modelo a representar todos os cenários possíveis e não apenas alguns selecionados.

Novamente reportando ao exemplo em questão, os dados coletados sobre a fábrica provavelmente serão registrados como um conjunto de séries temporais, ou seja, uma sequência de leituras coletadas periodicamente. Por exemplo, podemos ter um registro da temperatura a cada minuto, a taxa de produção a cada hora e o nível de vibração segundo a segundo. Após coletarmos os dados, precisaremos transformar essas séries temporais em algo mais apropriado para a construção do modelo.

C.2.2.5 Classificar os dados

Esta etapa determina quais dados representam a fase de operação “normal” e “anormal”. Estas informações devem ser fornecidas durante o processo de treinamento para que o sistema aprenda a classificar as entradas. O processo de associar dados a classes é chamado de rotulagem. As classes “normais” e “anormais” são, por exemplo, os rótulos.

Esse tipo de treinamento, no qual se instrui o algoritmo sobre o significado dos dados, é denominado de aprendizado supervisionado. O modelo de classificação resultante deste treinamento será capaz de processar os dados recebidos e prever a qual classe eles provavelmente pertencem.

Voltando ao exemplo aqui abordado, para poder rotular os dados de séries temporais coletadas, é preciso registrar quais períodos de tempo a máquina estava funcionando e quais períodos de tempo ela estava parada devido a um problema. É bastante razoável supor que o período imediatamente anterior à quebra da máquina geralmente representa uma operação anormal. No entanto, como não se pode necessariamente identificar uma operação anormal a partir de uma análise superficial dos dados, obter esta informação de forma mais corretamente possível requer alguma experimentação.

Após a tomada da decisão de como rotular os dados, pode-se então gerar uma série temporal que contenha os rótulos e adicioná-la ao conjunto de dados.

Exemplos de intervalo de fontes de dados são apresentados na Tabela 1. Por exemplo, a temperatura é registrada uma vez por minuto. Também geramos uma

série temporal que contém os rótulos dos dados. Neste exemplo o intervalo para os rótulos é 1 por 10 segundos, que é o mesmo que o menor intervalo para as outras séries temporais. Isso significa que podemos determinar um rótulo para cada ponto de dados em nossos dados, ou seja, se cada ponto é “normal” ou “anormal”.

Tabela 3 – Exemplo de fontes de dados, unidades e rótulos

Fonte de dados	Intervalo	Leitura de uma amostra
Taxa de produção	Uma vez a cada 2 minutos	100 unidades
Temperatura	Uma vez por minuto	30℃
Vibração	Uma vez a cada 10 segundos	23%
Rótulo ("normal"ou "anormal")	Uma vez a cada 10 segundos	normal

Fonte: Adaptado de WARDEN, 2019, pg. 15

Com os dados coletados e o rótulo estabelecido, o próximo passo é usá-los para projetar e treinar um modelo.

C.2.2.6 Projetar a arquitetura do modelo

No aprendizado profundo há vários tipos de arquiteturas já estudadas e aplicadas a diferentes contextos. Estas arquiteturas são projetadas para resolver uma ampla gama de problemas. Ao treinar um modelo, pode-se tomar dois caminhos: projetar sua própria arquitetura ou baseá-la em uma arquitetura existente desenvolvida por pesquisadores. Para muitos problemas comuns, é possível encontrar modelos pré-treinados disponíveis online, muitas vezes gratuitamente.

No decorrer deste trabalho, são apresentadas algumas arquiteturas de modelo, mas há muitas possibilidades além do que é apresentado aqui. Projetar um modelo é uma arte e uma ciência. A arquitetura do modelo é uma área importante de pesquisa onde novas arquiteturas são inventadas todos os dias.

O ponto de partida para definir qual arquitetura de modelo utilizar está intrinsecamente relacionado ao tipo de problema a resolver, o tipo de dados aos quais se tem acesso e as maneiras pelas quais se pode formatar e condicionar esses dados antes de inseri-los na como entrada em um modelo (este processo também é chamado de transformação). O aspecto fundamental a ser observado é que a arquitetura de modelo mais eficaz está ligada ao o tipo de dados com os quais se está trabalhando, ou seja, em outras palavras, os dados e a arquitetura de seu modelo estão fortemente interligados. Por questões didáticas ambos os tópicos serão tratados em separado, mas ambos sempre devem ser considerados em conjunto.

Um fator importante na fase de projeto é ponderar sobre as restrições do dispositivo que executará o modelo, uma vez que os microcontroladores, em geral,

têm memória limitada e processadores relativamente lentos. Modelos mais sofisticados e maiores exigem mais quantidade de memória disponível e demandam maior capacidade de processamento. Pode-se pensar que o tamanho de um modelo está estritamente relacionado ao número de neurônios que ele contém e a maneira como esses neurônios estão conectados. Um ponto importante a considerar é que alguns dispositivos incorporam mecanismos de aceleração de hardware, que pode beneficiar a execução de certos tipos de arquiteturas de modelo, portanto, é bastante desejável adaptar seu modelo aos pontos fortes do dispositivo que se deseja utilizar.

Neste trabalho é proposto começar treinando um modelo simples com apenas algumas camadas de neurônios e depois refinar a arquitetura em um processo iterativo até obter um resultado útil.

Os modelos de aprendizado profundo possuem, em geral, uma estrutura especial para os dados de entrada e suas saídas, chamadas de tensores. Considerando o contexto deste trabalho pode-se definir um tensor como uma lista que pode conter números ou outros tensores, ou seja, um tensor tem a forma de uma matriz. O modelo hipotético utilizado neste trabalho recebe um tensor como entrada. Na próxima seção explora-se com mais profundidade o conceito de tensor e de como aplicá-lo.

C.2.2.7 Terminologia a ser utilizada: tensores

A estrutura de um tensor é conhecida como sua forma e esta estrutura, em geral, apresenta várias dimensões. Exemplos de estruturas possíveis para a criação de um tensor são apresentados na Tabela 2.

Tabela 4 – Estruturas possíveis para a criação de um tensor

Vetor	[42 35 8 643 7]	Este é um exemplo de um tensor com uma única dimensão (um tensor 1D). Trata-se de um vetor de forma (5,) porque contém cinco números em uma única dimensão.
Matriz	[[123] [456] [789]]	Uma matriz é um tensor 2D, semelhante a um array 2D. Esta matriz, por exemplo, tem forma (3, 3) porque contém três vetores de três números.
Tensor de ordem maior	[[[123] [456] [789]] [[123] [456] [789]]]	Qualquer forma com mais de duas dimensões é chamada apenas de tensor. Aqui está um tensor 3D que tem forma (2, 3, 3) porque contém duas matrizes de forma (3, 3).
Escalar	43	Um número único, um escalar, é tecnicamente um tensor de dimensão zero. Por exemplo, o número 42 é um escalar e um tensor de dimensão zero.

Fonte: Adaptado de WARDEN, 2019, pg. 16

O modelo a ser usado neste trabalho é definido para aceitar algum tipo de tensor como entrada, todavia, o exemplo que vem sido utilizado apresenta dados na forma de séries temporais. A questão a ser resolvida nesta seção é como transformamos esses dados de séries temporais em um tensor que podemos passar como entrada para o modelo.

O primeiro passo nesta direção é decidir como gerar atributos a partir dos dados. É importante frisar que, no aprendizado de máquina, o termo "atributo", ou do inglês *"feature"*, refere-se a um tipo específico de informação na qual um modelo é treinado. Diferentes tipos de modelos são treinados com diferentes tipos de atributos. Por exemplo, um modelo pode aceitar um único valor escalar como seu único recurso de entrada.

Em geral as entradas de um modelo podem ser muito mais complexas do que apenas um escalar. Um modelo projetado para processar imagens pode aceitar um tensor multidimensional de dados de imagem como sua entrada, e um modelo projetado para realizar previsões com base em vários atributos pode aceitar um vetor contendo vários valores escalares, um para cada característica. Assim como um modelo pode aceitar apenas um valor escalar como seu único atributo de entrada.

Vale lembrar que a decisão neste trabalho, e no exemplo adotado, foi que o modelo deve usar a taxa de produção, a temperatura e a vibração para fazer suas previsões. Estes dados, na sua forma bruta, como séries temporais com intervalos de tempo diferentes, não são adequados para passar serem utilizados como entrada para o modelo. Algo que é abordado no próximo capítulo.

C.2.2.8 Definir o janelamento

Na figura 1 estão apresentados as séries de dados e a respectiva janela de tempo no qual cada dado é coletado. Nesta figura, cada asterisco corresponde a um dado coletado. O rótulo é incluído nos dados, pois o rótulo é necessário para o treinamento do modelo. O objetivo é treinar um modelo que possa ser usado para prever se a máquina está operando normalmente ou anormalmente em um determinado momento com base nas condições daquele momento. Note que o rótulo deve possuir uma amostragem igual à menor amostragem dentre as séries de dados.

Figura 27 – Séries de dados

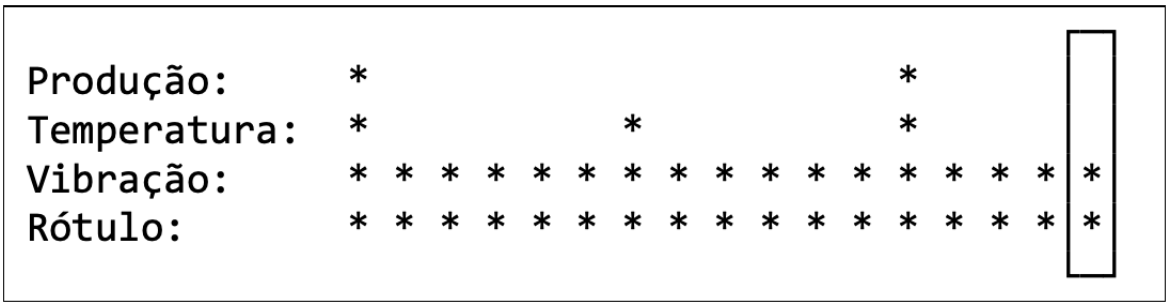
Produção	* *	(a cada 2 minutos)
Temperatura	* * *	(a cada minuto)
Vibração	* * * * * * * * * * * * * * * * * * * *	(a cada 10 segundos)
Rótulo	* * * * * * * * * * * * * * * * * * * *	(a cada 10 segundos)

Fonte: WARDEN, 2019, pg. 17

Observa-se que as séries temporais têm intervalos diferentes, se for passado

como entrada para o modelo apenas os dados disponíveis em um determinado instante de tempo, talvez estes dados não incluam todos os tipos de dados disponíveis. Por exemplo, no momento destacado na figura 2, apenas a vibração está disponível no instante de tempo tomado como amostra. Isso significa que o modelo só receberia informações sobre vibração para realizar a previsão (inferência).

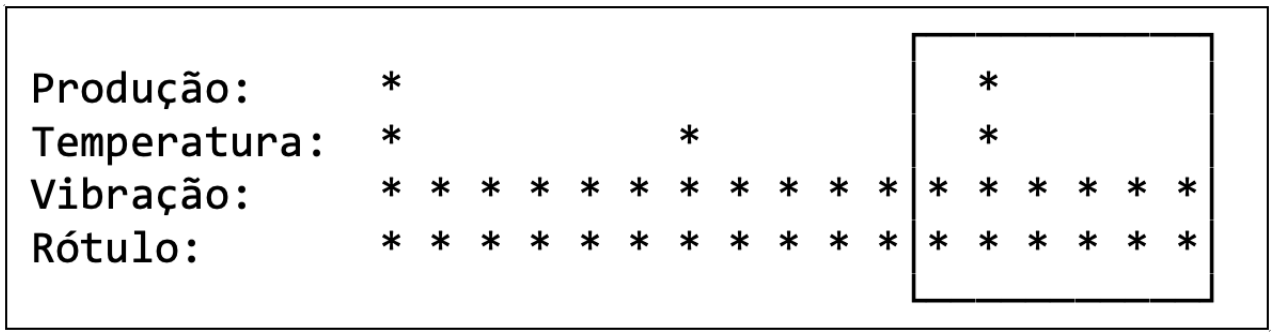
Figura 28 – Diferentes intervalos com diferentes amostragems



Fonte: WARDEN, 2019, pg. 17

A solução para esse problema é escolher uma janela no tempo que combine todos os dados em um único conjunto de valores. Por exemplo, podemos definir uma janela de um minuto e examinar todos os valores contidos nela. A figura 3 ilustra este exemplo.

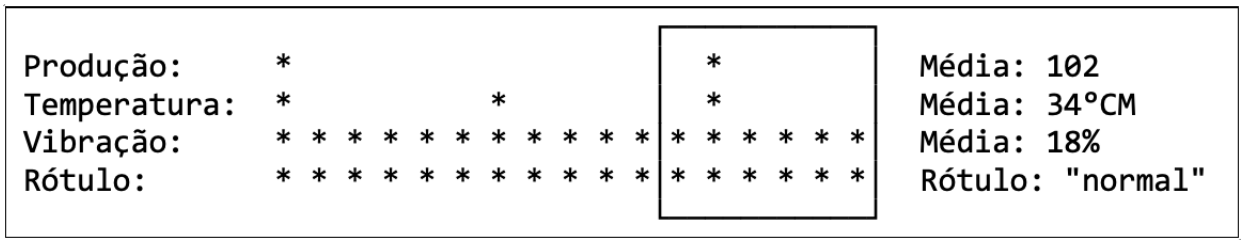
Figura 29 – Aumento da janela para cobrir todas as entradas



Fonte: WARDEN, 2019, pg. 17

Outra alternativa de solução possível seria realizar a média de todos os valores na janela adotada para cada série temporal e, para as séries que não permitam calcular a média, selecionar o valor mais recente. Desta forma, têm-se um conjunto de valores únicos, para cada janela de tempo. Em seguida é possível decidir como rotular essa janela de tempo baseado na presença de rótulos “anormais”. Se houver algum "anormal"presente, a janela deve ser rotulada como "anormal". Caso contrário, esta janela de tempo deve ser rotulado como “normal”. Na figura 4 mostra-se um exemplo de rotulação “normal”.

Figura 30 – Exemplo de rotulação normal



Fonte: WARDEN, 2019, pg. 17

Os três valores não rotulados são denominados como atributos. Deve-se passá-los para o modelo como um vetor, com um elemento para cada série temporal. No exemplo da figura 4 o vetor é: [102 34 0.18].

Durante o treinamento do modelo, pode-se obter uma nova janela a cada 1 minuto e passá-la para o modelo, usando o rótulo para informar ao algoritmo de treinamento do modelo se esta janela contém ou não alguma anormalidade.

Durante o processo de inferência (previsão da situação da máquina a partir de uma entrada), sempre que se usar o modelo para prever um comportamento anormal, pode-se apenas examinar os dados, calcular a janela mais recente, inseri-la como entrada no modelo e receber uma previsão.

Essa é uma abordagem simplista e pode nem sempre funcionar na prática, mas é um bom ponto de partida. Na verdade, sabe-se que o aprendizado de máquina é uma questão de tentativa e erro.

É importante examinar um último aspecto sobre os valores de entrada, antes de prosseguir para o treinamento do modelo.

C.2.2.9 Normalizar a representação

Na maior parte das aplicações de redes neurais os dados de entrada estarão na forma de tensores preenchidos com valores representados em números com ponto flutuante (*float*). Para que o algoritmo de treinamento funcione efetivamente, esses valores em ponto flutuante precisam ser semelhantes em tamanho. Na verdade, é ideal que todos os valores sejam expressos como números no intervalo de 0 a 1.

Revisitando o tensor de entrada do exemplo que vimos usando, temos: [102 34 0.18]. Estes números estão em escalas muito diferentes: a temperatura é maior que 100, enquanto a vibração é expressa como uma fração de 1. Para passar esses valores para a rede neural (o modelo), é necessário normalizá-los para que fiquem todos em uma faixa semelhante.

Uma maneira de fazer isso é calcular a média de uma série típica no conjunto de dados e subtraí-la dos valores. Isso tem o efeito de comprimir os números para que

fiquem mais próximos de zero. Um exemplo deste procedimento está representado na figura 5.

Figura 31 – Exemplo de normalização usando médias

```
#Séries de temperatura:
[108 104 102 103 102]
#Média:
103.8
#Valores normalizados, calculados subtraindo 103.8 de cada temperatura:
[ 4.2 0.2 -1.8 -0.8 -1.8 ]”
```

Fonte: WARDEN, 2019, pg. 18

Um muito comum em aprendizado de máquina é o tratamento de imagens. Uma situação encontrada com frequência, e na qual a normalização é implementada, é quando as imagens são processadas em máquinas digitais (computadores). Os computadores geralmente armazenam imagens como matrizes usando inteiros de 8 bits, cujos valores variam de 0 a 255. Para normalizar esses valores de forma que fiquem todos entre 0 e 1, cada valor de 8 bits é multiplicado por 1/255.

C.2.3 Utilizando o aprendizado de máquina

No exemplo da fábrica, os próximos passos para a geração do modelo seriam: definição de um objetivo adequado, coletar e rotular os dados apropriadamente, projetar os atributos que serão passados para o modelo e escolher uma arquitetura de modelo. Não importa qual problema a ser abordado, este usará esta mesma abordagem. É importante observar que esse é um processo iterativo, e talvez seja necessário passar muitas vezes por todos estágios do trabalho até chegar a um modelo que funcione.

Uma aplicação bastante usual que uso aprendizado de máquina é a construção de um modelo para prever o clima. É preciso decidir sobre o objetivo (por exemplo, prever se vai chover amanhã), coletar e rotular um conjunto de dados (como boletins meteorológicos dos últimos anos), projetar os atributos que alimentarão o modelo (talvez as condições médias dos últimos dois dias), escolher uma arquitetura de modelo adequada para esses tipos de dados e, por fim, definir o dispositivo no qual o modelo deve ser executado. Desta forma, o caminho a seguir é pensar em algumas alternativas iniciais, testar estas alternativas e ajustar a abordagem até se obter bons resultados, ou seja, um processo iterativo.

C.2.3.1 Treinar o modelo

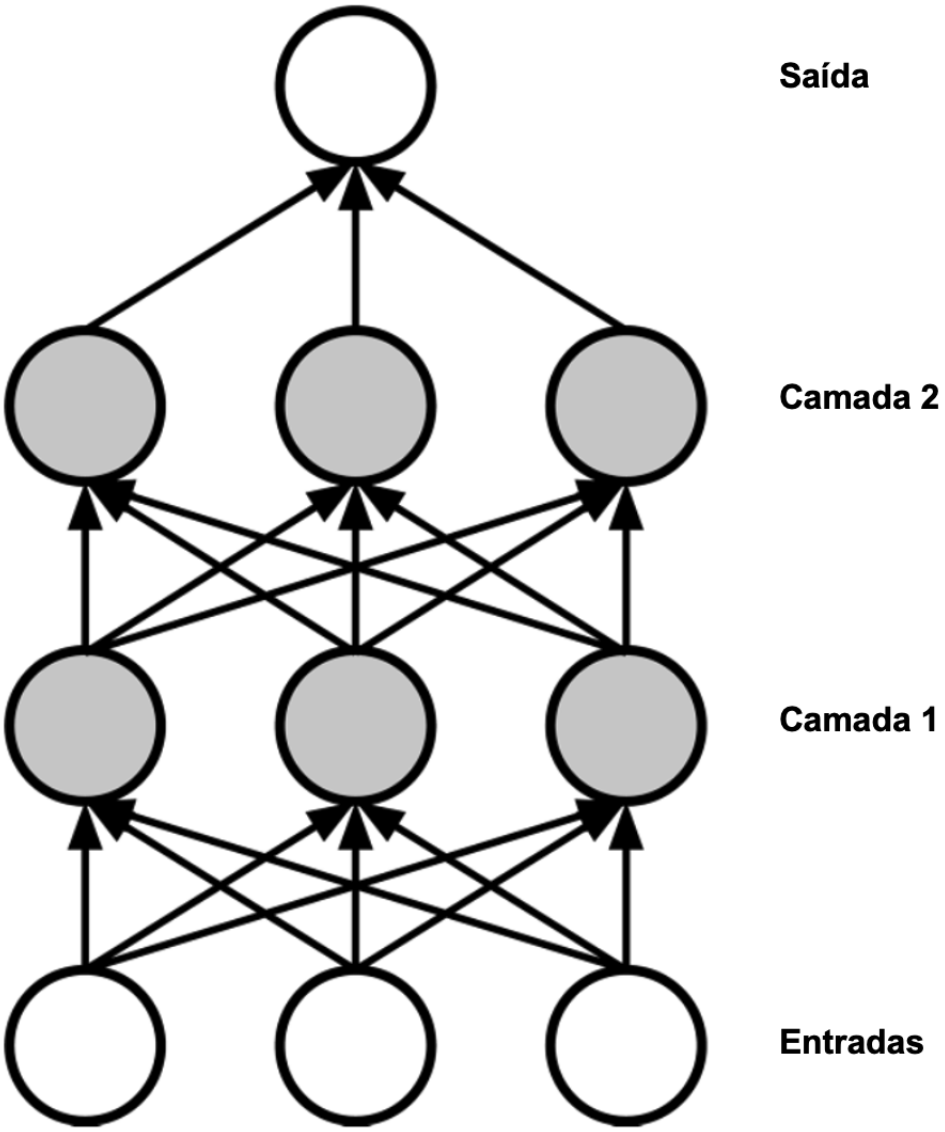
O treinamento é o processo pelo qual um modelo aprende a produzir a saída correta para um determinado conjunto de entradas. Consiste em inserir entradas de

dados em um modelo e fazer pequenos ajustes no modelo até que ele faça as previsões mais precisas possíveis. O peso é geralmente multiplicado pela entrada e o somatório das entradas, multiplicadas cada uma pelos seus pesos pode ser somado a um viés opcional antes de servir como entrada para um próximo neurônio artificial.

Conforme já apresentado, um modelo é uma rede de neurônios simulados (neurônios virtuais) representados por matrizes de números dispostos em camadas. Esses números são conhecidos como pesos e vieses *(biases)*, ou de um ponto de vista mais geral, como os parâmetros da rede.

Quando os dados são inseridos na rede neural eles são transformados por sucessivas operações matemáticas que envolvem os pesos e vieses de cada camada. A saída do modelo é o resultado da execução da entrada por meio dessas operações. Como exemplo, a figura 6 mostra uma rede simples de duas camadas.

Figura 32 – Rede neural simples de duas camadas



Fonte: WARDEN, 2019, pg. 19

Os pesos do modelo começam com valores aleatórios e os vieses geralmente começam com um valor de 0. Durante o treinamento, sequeências de dados são inseridos no modelo e a saída do modelo é comparada com a saída desejada (que no exemplo que está sendo usado neste trabalho é rótulo “normal” ou “anormal”).

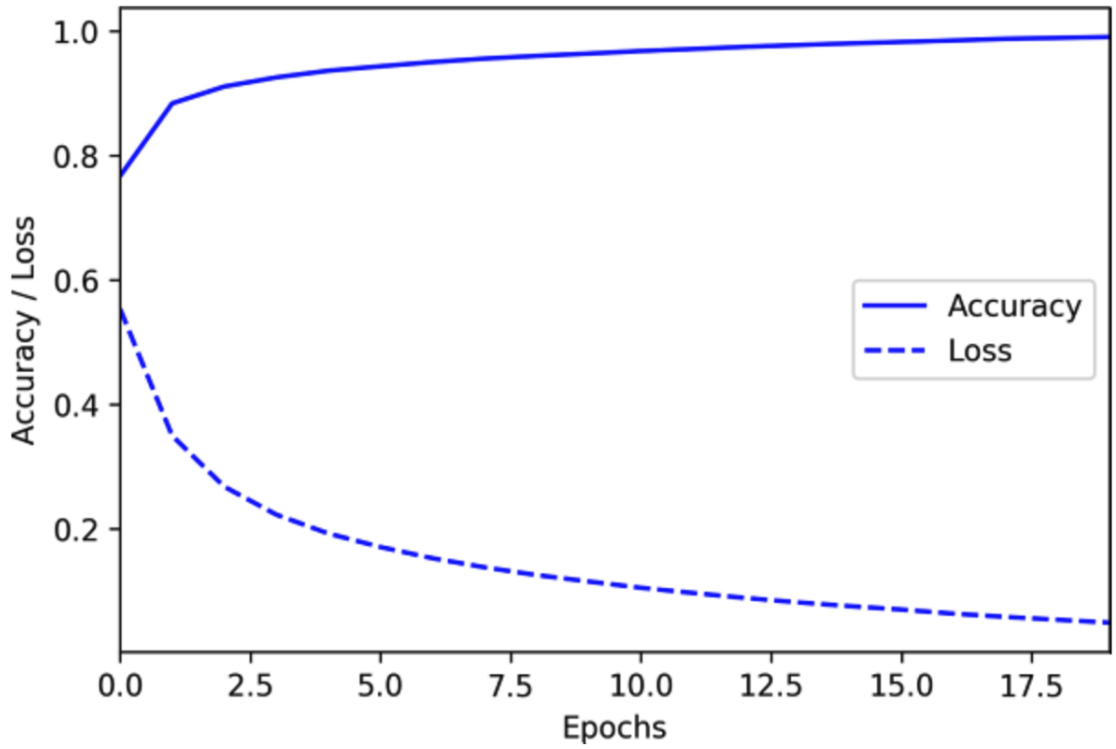
Um algoritmo chamado *back-propagation* ajusta os pesos e vieses de forma incremental para que, com o passar do treinamento, a saída do modelo se aproxime do valor desejado. O treinamento, que é medido em épocas (ou seja, iterações), continua até que se decida parar.

Como regra geral, o treinamento é finalizado quando o desempenho de um modelo não melhora mais, ou melhora de forma muito pouco significativa. A partir do momento em que se começa a fazer previsões precisas e exatas, diz-se que o processo convergiu.

Uma das maneiras usadas para determinar se um modelo convergiu, é a análise gráfica de seu desempenho durante o treinamento. Duas métricas de desempenho comumente utilizadas são a perda *(loss)* e a precisão *(accuracy)*. A métrica de perda fornece uma estimativa numérica de quão longe o modelo está de produzir as respostas esperadas, e a métrica de precisão fornece o percentual do tempo que o modelo escolhe a previsão correta. Um modelo perfeito teria uma perda de 0 e uma precisão de 100%, mas os modelos reais raramente atingirão estes patamares.

A figura 7 ilustra o processo, ou seja, mostra a evolução das perdas e da precisão durante o processo de treinamento para uma rede que usa aprendizado profundo. Com base em gráficos deste tipo é possível verificar, na medida em que o treinamento avança, como a precisão aumenta e as perdas são reduzidas, até que se atinja um ponto em que o modelo não melhora mais, ou melhora muito pouco.

Figura 33 – Convergência do modelo durante treinamento



Fonte: WARDEN, 2019, pg. 20

Caso a convergência apresente-se difícil de obter, é possível melhorar o desempenho do modelo, alterando e ajustando vários valores usados na configuração do modelo, além de se ter o recurso de moderar o processo de treinamento. Esses valores usados na configuração são conhecidos, de uma ótica mais geral, como hiperparâmetros e incluem variáveis, tais como o número de períodos de treinamento a serem executados e o número de neurônios em cada camada. A cada alteração realizada no modelo, em geral, deve-se treiná-lo novamente, examinar as métricas e decidir se é necessário otimizá-lo ainda mais. Espera-se, com esta metodologia, que o resultado seja um modelo com precisão, pelo menos, aceitável.

Mesmo seguindo esta metodologia e persistindo em buscar uma melhora no modelo, é importante frisar que não há garantia de obter-se a precisão suficiente para o problema que se está tentando resolver. Nem sempre há informações suficientes no conjunto de dados disponíveis e usados para fazer as previsões de forma precisa e, é importante saber, que alguns problemas simplesmente não podem ser resolvidos, mesmo com o aprendizado profundo de última geração. Sendo assim, mesmo que não se obtenha um modelo 100% preciso, ele pode ser útil, dependendo do contexto ao qual ele é submetido. No exemplo da máquina fabril, se ele for capaz de prever uma operação anormal, mesmo que apenas algumas vezes, este modelo ainda pode ter um grande valor.

C.2.3.2 Sub-ajuste e super-ajuste

Há duas principais razões pelas quais um modelo não converge e elas são denominadas modelo sub-ajustado *(underfitting)* e super-ajustado *(overfitting)*. Os termos originais em inglês foram aqui mencionados, pois, em geral, são usados por melhor representarem do ponto de vista semântico os significados.

O foco principal de uma rede neural é aprender a ajustar seu comportamento aos padrões que reconhece nos dados. Diz-se que um modelo está corretamente ajustado, se ele produz a saída correta para um determinado conjunto de entradas. Quando um modelo está sub-ajustado, ele ainda não foi capaz de aprender suficientemente os padrões da entrada para poder fazer boas previsões. Isso pode acontecer por vários motivos, entretanto o mais comum deve-se a arquitetura. Por que é muito pequena para capturar a complexidade do sistema que deve modelar, ou porque não foi treinada suficientemente.

Por outro lado, quando um modelo está super-ajustado, é capaz de prever com exatidão, incluindo as minúcias de seus dados de treinamento, mas não é capaz de generalizar seu aprendizado para dados os quais não foram submetidos anteriormente. Muitas vezes isso acontece porque o modelo conseguiu aprender inteiramente os dados utilizados no treinamento ou então aprendeu a confiar em um atalho presente nos dados de treinamento, mas não no mundo real.

Para exemplificar esta situação, que pode não parecer tão clara, toma-se um exemplo. Suponha-se que se está treinando um modelo para classificar fotos que contenham cães ou gatos. Se todas as fotos de cães usadas como dados de treinamento forem tiradas ao ar livre e todas as fotos de gatos forem tiradas em ambientes fechados, o modelo pode aprender, de forma errônea, a usar a presença do céu em cada fotografia para prever qual animal está presente na foto. Isso significa que pode classificar incorretamente futuras fotos de cães se forem tiradas em ambientes fechados.

Há várias maneiras de evitar e prevenir o super-ajuste. Uma possibilidade é reduzir o tamanho do modelo. Isso fará com que ele não tenha capacidade suficiente para aprender uma representação exata de seu conjunto de treinamento. Um conjunto de técnicas conhecidas como *regularização* pode ser aplicado durante o treinamento para reduzir o grau de super-ajuste. Para aproveitar ao máximo os dados disponíveis, se forem limitados, uma técnica chamada aumento de dados *(data augmentation)* pode ser usada. Esta técnica consiste em gerar novos dados artificiais, dividindo e sub-dividindo os dados existentes. Mas a melhor maneira de evitar o super-ajuste, é buscar um conjunto de dados maior e, principalmente, mais variado (quanto mais, melhor).

C.2.3.3 Regularização e aumento do conjunto de dados

As técnicas de regularização devem aplicadas nos modelos de aprendizado profundo que sejam mais propensos ao fenômeno do super-ajuste. Estas técnicas, em geral, envolvem restringir o modelo de alguma forma para evitar que ele memorize perfeitamente os dados que são usados durante o processo de treinamento.

Há vários métodos que podem ser usados para a regularização. Alguns, como as denominadas regularizações L1 e L2, envolvem ajustes nos algoritmos usados durante o treinamento para penalizar modelos complexos que são propensos a super-ajuste. Outro método, chamado *dropout*, envolve o corte aleatório das conexões entre os neurônios durante o treinamento.

C.2.4 Testar, solucionar problemas e validar o modelo

Uma das formas de avaliar o desempenho de um modelo, é verificar como ele se comporta em relação aos dados de entrada no processo de treinamento. No entanto, isso não é suficiente, pois permite fazer apenas uma validação parcial. Durante o treinamento, um modelo aprende a ajustar sua resposta aos dados de treinamento, da melhor forma. Como discutido anteriormente, em alguns casos o modelo começará a super-ajustar os dados de treinamento, o que significa que funcionará muito bem com dados de treinamento, mas não quando for submetido a novos dados.

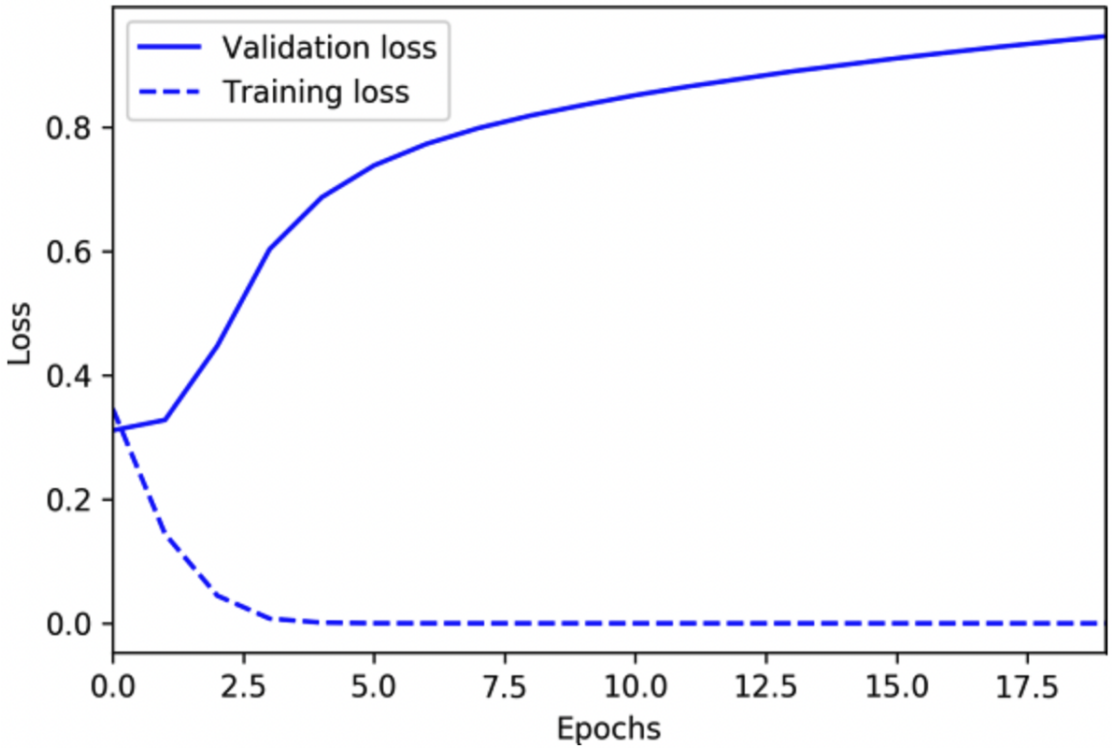
Para detectar quando isso está acontecendo, é preciso validar o modelo usando novos dados, que não foram usados no treinamento. Um método eficaz é dividir um conjunto de dados em três partes: treinamento, validação e teste. Uma divisão típica é usar 60% para dados de treinamento, 20% para validação e 20% para teste. Um cuidado essencial para essa divisão, é que ela deve ser feita de forma que cada parte contenha a mesma distribuição de informações de maneira a preservar a estrutura dos dados. Por exemplo, como os dados do exemplo usado neste trabalho são uma série temporal, é possível dividi-los em três partes contíguas de tempo. Se os dados não fossem uma série temporal, seria possível apenas amostrar os pontos de dados aleatoriamente.

Uma possível estratégia parra validar o modelo é, durante o treinamento, usar um determinado conjunto de dados para treinar o modelo. Periodicamente, os dados referentes ao conjunto de dados de validação são inseridos no modelo e a perda é calculada. Como o modelo não recebeu esses dados anteriormente, a sua pontuação de perda é a medida mais confiável do desempenho do modelo. Ao comparar a perda no processo de treinamento com a perda no processo de validação (e precisão, ou quaisquer outras métricas disponíveis) ao longo do tempo, é possível verificar se o modelo está super-ajustado.

A figura 8 apresenta o comportamento de um modelo com super-ajuste.

No gráfico é possível verificar como a perda durante o treinamento diminuiu, mas a perda durante o processo de validação aumentou. Isso significa que o modelo está melhorando na previsão quando os dados de treinamento são usados, mas está perdendo sua capacidade de generalizar quando novos dados são inseridos.

Figura 34 – Gráfico com modelo super-ajustado durante o processo de treinamento



Fonte: WARDEN, 2019, pg. 21

Na medida em que são realizados ajustes nos modelos durante o processo de treinamento com o objetivo de melhorar o desempenho do modelo e evitar o super-ajuste, espera-se que as métricas de validação também apresentem melhoria.

No entanto, esse procedimento tem um efeito colateral indesejável. Ao buscarmos a melhoria das métricas de validação, pode-se estar apenas acelerando o caminho para o super-ajuste. Cada alteração que se faz levará o modelo a um melhor ajuste aos dados de validação, e, no final, pode-se ter o mesmo problema de super-ajuste.

Todavia é possível adotar um mecanismo para verificar se isso não aconteceu. Na etapa final do processo de treinar um modelo insere-se os dados de teste para confirmar se o modelo funciona tão bem quanto durante a validação. Caso isto não ocorra, é por que o modelo foi para o super-ajuste no processo de treinamento e validação. Nesse caso, talvez seja necessário voltar ao início e criar uma nova arquitetura de modelo, pois, ao continuar a fazer ajustes para melhorar o desempenho do atual modelo, muito provavelmente ocorrerá a intensificação do super-ajuste.

O processo de treinamento pode ser considerado satisfatório e concluído depois que se tiver um modelo com desempenho aceitável quando submetido aos dados de treinamento, validação e teste. A próxima etapa é preparar o modelo para ser executado no dispositivo no qual ele será usado. Nesta etapa é que o TensorFlow Lite se mostra útil, pois com ele é possível converter modelos e executá-los em dispositivos compatível com a linguagem C++.

Após converter o modelo usando o TensorFlow Lite pode-se executar uma verificação de inferência antes mesmo de embarcar o mesmo no dispositivo. Isto é importante pois é possível verificar o comportamento do modelo, visando observar se houve perda de confiança. Finalmente, é possível incorporar o modelo no microcontrolador e executar o mesmo utilizando a biblioteca TensorFlow Lite para Microcontroladores.