

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

Lucas Martins Mendes

**Estudo e Avaliação da Plataforma Computacional Embarcada NVIDIA Jetson
NANO para Aplicações de Redes Neurais Convolucionais na borda**

FLORIANÓPOLIS, 2022.1

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

Lucas Martins Mendes

**Estudo e Avaliação da Plataforma Computacional Embarcada NVIDIA Jetson
NANO para Aplicações de Redes Neurais Convolucionais na borda**

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 Engenharia Mecatrônica.

Orientador:
Prof. MAURÍCIO EDGAR STIVANELLO, DR.
ENG.

FLORIANÓPOLIS, 2022.1

Ficha de identificação da obra elaborada pelo autor.

Mendes, Lucas

**Estudo e Avaliação da Plataforma Computacional Embarcada
NVIDIA Jetson NANO para Aplicações de Redes Neurais Convolucionais
na borda / Lucas Mendes; orientação
de Maurício Edgar Stivanello. – Florianópolis, SC,
2022.**

58 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. **Aprendizado de Máquina.** 2. **Redes Neurais Convolucionais.**
3. **Sistemas Embarcados.** 4. **Visão Computacional.**
5. **NVIDIA Jetson NANO.** I. Edgar Stivanello, Maurício.
II. Instituto Federal de Santa Catarina. III. **Estudo
e Avaliação da Plataforma Computacional Embarcada
NVIDIA Jetson NANO para Aplicações de Redes Neurais Convolucionais**

ESTUDO E AVALIAÇÃO DA PLATAFORMA COMPUTACIONAL EMBARCADA NVIDIA JETSON NANO PARA APLICAÇÕES DE REDES NEURAIS CONVOLUCIONAIS NA BORDA

LUCAS MARTINS MENDES

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

Florianópolis, 8 de Agosto, 2022.

Banca Examinadora:

Documento assinado digitalmente
 **Maurício Edgar Stivanello**
Data: 16/08/2022 14:59:12-0300
Verifique em <https://verificador.iti.br>

Prof. Maurício Edgar Stivanello, Dr. Eng.

Documento assinado digitalmente
 **ADRIANO REGIS**
Data: 15/08/2022 16:41:39-0300
Verifique em <https://verificador.iti.br>

Prof. Adriano Regis, Me.

IFSC

Documento assinado digitalmente
 **DELICINO PICININ JUNIOR**
Data: 16/08/2022 13:35:52-0300
Verifique em <https://verificador.iti.br>

Prof. Delcino Picinin Júnior, Dr.

IFSC

Ao meu Avô, Aristides Martins,
por sempre instigar e incentivar
minha incessante curiosidade sobre tudo.

AGRADECIMENTOS

Inicio agradecendo a Deus, que é meu norte e tem me guiado e guardado durante toda essa jornada deste curso, é o começo da minha caminhada como engenheiro.

Agradeço aos meus pais, Rodrigo Mendes e Rita Helena, pela incentivo, suporte e a paciência durante todo este processo. Eu não estaria aqui sem seu amor e sua atenção.

Ao meu irmão, Ian, um amigo sempre presente e companheiro para vida, pela paciência, compreensão e fraternidade.

Aos meus avós, Nelcy e Vilson (*in memorian*), muita gratidão pelo seu carinho e cuidado. À minha Avó Vania, que sempre investiu em tempo de qualidade e me apoiou da melhor forma que possível. Ao meu “Vô Ari”, um muito obrigado por me “apresentar” a física e a matemática. Às minhas tias Thais e Ludmila, pela convivência amorosa ao longo da minha vida e, também, pelas muitas correções durante o processo de elaboração deste documento. Agradeço à Ester e ao Danilo, meus tios, pela recepção calorosa e pelas conversas sobre tudo e nada, que me ajudaram tanto, em momentos difíceis. Agradeço a toda minha família.

Agradeço ao meu professor e orientador Maurício Edgar Stivanello, pelas inúmeras oportunidades de pesquisa e crescimento profissional e acadêmico durante esta jornada. Não poderia ter tido um coordenador e orientador melhor; que por acreditar no meu potencial me direcionou à minha área de trabalho atual. Obrigado por ter sido um ponto de referência, mesmo durante este período conturbado no mundo - pandemia, ensino remoto, etc - e em minha vida.

Agradeço às pessoas da Macnica DHW, a empresa onde trabalho, por valorizarem e darem o suporte necessário à minha formação e o processo acadêmico em geral.

Agradeço ao professor Delcino, da engenharia Mecatrônica, por elaborar e manter o *template* L^AT_EX utilizado neste documento, que me salvou tanto tempo e esforço na elaboração deste.

Agradeço aos servidores e funcionários do IFSC, Câmpus Florianópolis, que desde 2013, no meu curso técnico fazem parte na minha formação e tem colaborado no meu desenvolvimento como profissional, pessoa e cidadão.

RESUMO

Este trabalho propõe avaliar e explorar a integração entre aplicações de aprendizado de máquina baseados em redes neurais convolucionais (RNC) e sistemas embarcados, no contexto da visão computacional e da metodologia da computação de borda e processamento de dados distribuído. De forma a poder se explorar adequadamente os tópicos levantados, serão abordados os conceitos, tecnologias e metodologias fundamentais à compreensão e aplicação de redes neurais artificiais, e como isto interage com as particularidades de sistemas embarcados. Será apresentados os fundamentos de redes neurais artificiais e RNC, quais seus subcomponentes e particularidades relevantes. Em seguida será descrito as principais plataformas de *hardware* utilizadas com RNCs e introduzido a plataforma selecionada para esta comparação, a NVIDIA Jetson NANO, além de outras relevantes também consideradas. Também será descrita a metodologia chamada de “computação em borda” e como ela complementa os conceitos previamente explorados. Para poder adequadamente comparar as plataformas escolhidas, serão caracterizados critérios como consumo de energia, tempo de execução, uso de memória, etc e sua relevância. Na implementação das aplicações teste, foi utilizado a biblioteca FastAI e a linguagem de programação Python para poder rapidamente desenvolver aplicações utilizando redes neurais convolucionais baseadas em arquiteturas bem documentadas. Em seguida, é apresentado os bancos de dados selecionados como ferramenta de comparação e suas características e composição. Finalmente, são expostos os resultados obtidos para cada banco de dados selecionado e em seguida estes são discutidos em relação aos critérios de avaliação levantados anteriormente. Pode-se concluir que, apesar da complexidade inerente da plataforma, é viável se utilizar ferramentas de aprendizado de máquina em conjunto de sistemas embarcados para atender demandas reais e que existe grande potencial a se explorar neste tipo de implementação.

Palavras-chave: Aprendizado de Máquina; Redes Neurais Convolucionais; Sistemas Embarcados; Visão Computacional; NVIDIA Jetson NANO.

ABSTRACT

This work proposes to evaluate and explore the integration between machine learning applications based on convolutional neural network (CNN) and embedded systems through the lens of computer vision and the edge computing methodology. In order to be able to adequately explore the previously brought up topics, will be addressed the main concepts, technologies and methodologies relevant to the understanding and application of artificial neural networks (ANN) and how it interacts with embedded systems. It will be presented the core concepts ANN's and CNN's, their subcomponents and relevant particularities. In sequence, will be enumerated the main relevant hardware platforms used with CNN's and introduced the selected embedded platform for this comparison, the NVIDIA Jetson NANO, besides the other relevant platforms also considered. Also, it will be outlined the computing methodology called "edge computing" and how it complements the previously brought up concepts. To be possible to adequately compare the selected platforms, will be characterized criteria such as power consumption, execution time, memory usage, etc and it's relevance for this application. In the implementation of the test applications, it was used the FastAI library and Python programming language to be able to rapidly develop applications using CNN's based on well documented architectures. In sequence, is presented the datasets selected as a comparison tool and their characteristics and composition. Finally, the obtained results for each selected dataset are displayed and in sequence they are discussed in relation to the aforementioned evaluation criteria raised previously. It can be concluded that, besides the inherent complexity associated with the platform, it is viable to use machine learning tools in conjunction with embedded systems to meet real demands and that there is great potential to be explored in this kind of implementation.

Keywords: Machine Learning; Convolutional Neural Networks; Embedded Systems; Computer Vision; NVIDIA Jetson NANO.

LISTA DE FIGURAS

Figura 1 – Arquitetura de um perceptron multicamada clássico	16
Figura 2 – Gráfico da função de ativação $\tanh(x)$	17
Figura 3 – Exemplo de uma arquitetura de rede neural convolucional, <i>alexnet</i>	18
Figura 4 – Exemplo de convolução de <i>kernel</i>	19
Figura 5 – Exemplo do operador sobel	20
Figura 6 – Comparação de antes e depois de uma imagem para <i>Max Pooling</i>	21
Figura 7 – Gráfico da função de ativação <i>ReLU</i>	21
Figura 8 – Diagrama da placa de desenvolvimento NVIDIA Jetson NANO.	25
Figura 9 – Arquitetura de um sistema com processamento de borda	26
Figura 10 – Arquitetura do sistema proposto no Artigo	28
Figura 11 – Implementação física do sistema proposto no Artigo	28
Figura 12 – Amostra de imagens do banco de dados <i>Oxford-IIIT Pet</i> , duas para cada categoria disponível	35
Figura 13 – Exemplos de imagens do banco de dados e da tampa sem o ROI	37
Figura 14 – Matrizes de confusão para cada arquitetura de RNC avaliada	40
Figura 15 – Matrizes de confusão para cada arquitetura de RNC avaliada	41
Figura 16 – Exemplo das estatísticas fornecidas pela ferramenta <i>jtop</i> durante o processo de inferência	43
Figura 17 – Placa NVIDIA Jetson NANO com <i>cooler</i> acoplado	44

LISTA DE TABELAS

Tabela 1 – Especificações do computador utilizado para comparação	33
Tabela 2 – Composição banco de dados <i>Oxford-IIIT Pet</i>	36
Tabela 3 – Distribuição dos dados	37
Tabela 4 – Tempos de execução - <i>Oxford-IIIT Pet</i>	41
Tabela 5 – Tempos de execução - Rebites	41

LISTA DE ABREVIATURAS E SIGLAS

CPU	<i>Central Processing Unit</i>
EOL	<i>End of Life</i>
GPU	<i>Graphics Processing Unit</i>
GUI	<i>Graphical User Interface</i>
IFSC	Instituto Federal de Santa Catarina
IoT	<i>Internet of Things</i>
PPA	<i>Personal Package Archive</i>
RNA	Rede Neural Artificial
RNC	Rede Neural Convolucional
ROI	<i>Region of Interest</i>
SSH	<i>Secure Shell</i>
TDP	<i>Thermal Dissipation Profile</i>

SUMÁRIO

1	INTRODUÇÃO	13
1.1	Contextualização	13
1.2	Objetivos	14
1.2.1	Objetivo geral	14
1.2.2	Objetivos específicos	14
2	FUNDAMENÇÃO TEÓRICA	15
2.1	Redes Neurais Artificiais	15
2.1.1	Descrição Geral de uma RNA Clássica	15
2.1.2	RNCs	17
2.1.2.1	<i>Convolução de Kernel</i>	19
2.1.2.2	<i>Max pooling</i>	20
2.1.2.3	<i>Local Response Normalization</i>	21
2.1.2.4	<i>Uso de recursos</i>	22
2.1.2.5	<i>Transfer Learning</i>	22
2.2	Plataformas computacionais	23
2.2.1	CPU	23
2.2.2	GPU	23
2.2.3	Placas Aceleradoras	24
2.2.4	Sistemas Embarcados	24
2.3	Computação em borda e aplicações de visão computacional	26
2.4	Trabalhos correlatos	29
3	METODOLOGIA	31
3.1	Caracterização dos critérios a serem avaliados nos testes	31
3.1.1	Potência/Consumo de energia	31
3.1.2	Tempo de execução	31
3.1.3	Uso de memória	32
3.1.4	Ventilação	32
3.1.5	Dimensão física	32
3.1.6	Custo Financeiro	32
3.2	Ferramentas Utilizadas	33
3.2.1	Plataformas Avaliadas	33
3.2.2	Processo de Inferência e Treinamento	34
3.3	Aplicações a serem avaliadas	34
3.3.1	<i>Oxford-IIIT Pet</i>	34
3.3.2	Inspeção de rebites	35
3.4	Processo de configuração do ambiente	36
3.4.1	NVIDIA Jetson NANO	37
3.4.1.1	<i>Geral</i>	37
3.4.1.2	<i>Ferramentas de AI</i>	38
3.4.2	PC	39
3.5	Resultados experimentais	40
3.5.1	<i>Oxford-IIIT Pet</i>	40
3.5.2	Inspeção de rebites	41
3.5.3	Discussão dos Resultados	42
4	CONCLUSÕES E TRABALHOS FUTUROS	46

REFERÊNCIAS	48
APÊNDICES	50
APÊNDICE A – CÓDIGO EXEMPLO PARA MAX POOLING	51
APÊNDICE B – <i>SCRIPT</i> DE TREINAMENTO REBITES	52
APÊNDICE C – <i>SCRIPT</i> DE TREINAMENTO <i>OXFORDII PET</i> . . .	55
ANEXOS	58
ANEXO A – PÁGINA DE DADOS GERAIS NVIDIA JETSON NANO	59

1 INTRODUÇÃO

1.1 Contextualização

Nos últimos anos, cada vez mais tem crescido a demanda por aplicações de inteligência artificial, sendo ela inicialmente atendida por soluções em nuvem através da internet, como exemplificado em Brignoli (2021), que utiliza um servidor com uma unidade de processamento gráfica, ou em Wei *et al.* (2018), que propõe uma arquitetura para inferência de RNCs baseada em FPGA hospedado na nuvem, ou ainda em Shu *et al.* (2018), que propõe um método para aceleração de inferência para RNCs baseado em *offloading* para recursos em nuvem. Na medida que as tecnologias para uso de aprendizado de máquina se tornam mais disponíveis, vem sendo viabilizado também o processamento local, independente da nuvem. Isto tem aberto novas possibilidades de aplicação onde normalmente era de difícil implementação utilizando as soluções existentes ou plataformas clássicas, tipo PC x86_64.

Neste contexto, empregar recursos descentralizados para o processamento de dados mais próximo dos dispositivos cliente, como *smartphones* e *tablets*, é um paradigma de computação ascendente, referido como Computação de Borda ou *Fog/Edge Computing* (HONG; VARGHESE, 2019, Adaptado)

Aplicações não necessariamente precisam ser implementadas em dispositivos embarcados, mas o uso dos mesmos permite que este sistema melhor atenda os requisitos de consumo de energia, espaço, etc. Um exemplo de demanda para a computação de borda seria a inspeção visual e detecção de defeitos em uma linha de produção industrial. Um sistema que atenda a estes requisitos funcionais poderia ser implementado de forma clássica por meio de algoritmos de visão, ou por meio de algoritmos de aprendizado de máquina. Em ambos os casos, se necessitaria de baixa latência no processo de inspeção, caso contrário reduziria a produtividade do processo. Este é apenas um dos requisitos não funcionais observados em diversas aplicações que dificultam ou impedem o uso de processamento em nuvem.

Além disso, este paradigma de computação pode ser aplicado de modo que, além de atender situações onde antes era impossível o uso de aprendizado de máquina, melhore a privacidade e confiabilidade de aplicações já existentes, além de outros benefícios.

Um exemplo comum disso é o princípio de operação de assistentes de voz atualmente, que muitas vezes somente executam localmente a detecção de uma palavra-chave, e todo resto da conversão de áudio para texto, em um servidor remoto. Ou seja, dependem de servidores remotos mesmo para executar ações simples onde não obrigatoriamente demandaria acessar a internet, o que reduz a confiabilidade do sistema, além de abrir brechas para invasão de privacidade.

Atualmente, existem diversas plataformas projetadas para este fim, computação de borda, desde mini computadores que utilizam x86_64 como *intelNUC*TM, soluções modulares que podem ser inseridas em quase qualquer outro sistema como Google Coral e Intel VPU, até sistemas embarcados direcionados a processamento de dados e imagens local, como NVIDIA Jetson NANO baseado na arquitetura de GPU's.

Com base no cenário apresentado, pretende-se investigar o estado atual, a usabilidade e características no que se refere às plataformas de embarcadas disponíveis e suas potencialidades.

1.2 Objetivos

1.2.1 Objetivo geral

Estudar e realizar testes conceituais com diferentes plataformas de *hardware* sobre sua viabilidade para processamento embarcado de aplicações de visão computacional empregando Redes Neurais Convolucionais.

1.2.2 Objetivos específicos

- A) Estudar trabalhos correlatos e conceitos importantes de aprendizado de máquina em aplicações de visão computacional;
- B) Selecionar e preparar diferentes plataformas de computação para a realização dos testes;
- C) Realizar testes conceituais utilizando diferentes aplicações de aprendizado de máquina e métricas adequadas;
- D) Documentar os resultados;

2 FUNDAMENÇÃO TEÓRICA

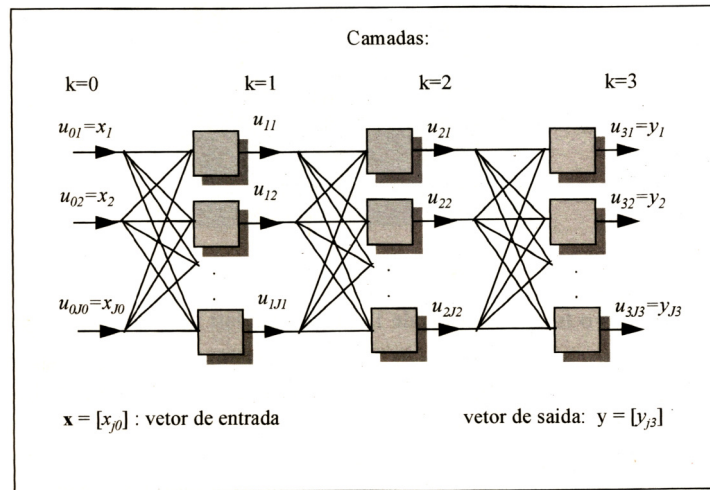
Nesta seção são explorados alguns tópicos essenciais ao entendimento do desenvolvimento realizado. Na primeira parte são apresentados conceitos gerais sobre redes neurais artificiais (RNA) e sobre o tipo utilizado especificamente no desenvolvimento dos testes conceituais envolvidos neste estudo. Em seguida são descritos conceitos relacionados às diferentes arquiteturas de computadores e sistemas de computação e como isso se relaciona com redes neurais artificiais. Subsequentemente são expostos conceitos sobre computação em borda e como isso se relaciona com os tópicos anteriores. Por último são explorados trabalhos correlatos existentes na literatura.

2.1 Redes Neurais Artificiais

Redes neurais artificiais (RNA) são uma forma encontrada de emular a capacidade de aprendizado e adaptação apresentado por seres vivos. Atualmente, estão disponíveis diversas implementações e arquiteturas de RNA, cada uma com suas características particulares. Nas seções a seguir serão descritas as arquiteturas necessárias à compreensão da avaliação elaborada neste trabalho.

2.1.1 Descrição Geral de uma RNA Clássica

Uma das primeiras arquiteturas de RNA criadas, desenvolvida por Rosenblatt no fim da década de 1950 (KOVACS, 1996), baseado no trabalho de Warren McCulloch (KOVACS, 1996 apud MCCULLOCK; PITTS, 1943) é denominada de discriminadores lineares ou *Perceptron*. Ela utiliza uma estrutura que distribui neurônios em camadas, onde cada um está conectado a todos os outros da camada seguinte, como ilustrado na figura 1.

Figura 1 – Arquitetura de um perceptron multicamada clássico

Fonte: (KOVACS, 1996)

Este tipo de rede possui ao menos 3 camadas:

1. uma de entrada, que está conectada a estímulos externos;
2. uma ou mais camadas ocultas;
3. uma de saída, que representa a interpretação da rede das entradas providas;

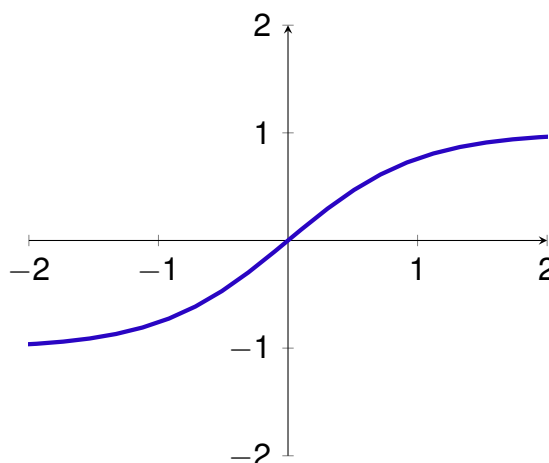
Cada conexão entre os neurônios possui fatores numéricos associados, chamados de pesos e desvio¹, que são aplicados às conexões de cada camada. Para obter seu valor de ativação, um neurônio executa uma soma de todos os seus valores de entrada multiplicados pelos seus respectivos pesos mais o desvio, como na equação 1.

$$u_{n+1} = m_{n+1} + \sum_{x=0}^n u_{nx} \cdot n_{nx} \quad (1)$$

sendo u_{n+1} o vetor de valores de entrada, n_n o vetor contendo os pesos daquele neurônio e m_{n+1} o seu desvio. Após computado a soma, o valor obtido é normalizado por meio de uma função matemática, usualmente algum tipo de função sigmóide, como $\tanh(x)$, demonstrada na figura 2, ou alguma outra função logística que mapeia suas entradas de forma que para qualquer $x \in \mathbb{R}$ sua saída cai dentro do intervalo $[-1, 1]$.

A primeira camada é utilizada para receber os valores de entrada da rede e a última camada representa os valores de saída, para os quais podem ser atribuídas interpretações diversas dependendo da aplicação final.

¹ Comumente chamados de *weights & bias*

Figura 2 – Gráfico da função de ativação $\tanh(x)$ 

Fonte: do autor

Além do perceptron, também existem diversas outras arquiteturas para redes neurais, sendo que a principal delas, para este trabalho, é a que é chamada de Rede Neural Convolutiva (RNC).

Como este trabalho trata de RNC's da perspectiva de um usuário, somente será descrito de forma breve sua arquitetura, e não será explorado certos detalhes de implementação e funcionamento, exceto quando relevante.

2.1.2 RNCs

Redes Neurais Convolucionais (RNCs²) são uma arquitetura de RNA baseada no princípio de convolução de *kernel*, um conceito muito comumente aplicado no campo de sistemas de visão e visão computacional. Esta arquitetura de rede neural é particularmente bem adaptada ao processamento de imagens, pois operam de modo a detectar características e agrupamentos das mesmas nas suas entradas. Como é estruturada em camadas, essas características são categorizadas hierarquicamente até a camada de saída, que pode representar alguma avaliação de alto nível sobre uma imagem, como no caso binário: 'contém um gato' ou 'não é uma casa', ou no categorizador: 'modelo de um carro', 'raça do cachorro', 'espécie de uma planta'.

Este tipo de arquitetura de rede neural também pode ser utilizada em diversas outras aplicações, como análise de linguagem natural (KIM, 2014), reconhecimento de voz (GRAVES; MOHAMED; HINTON, 2013), entre outras aplicações. Para o propósito deste trabalho somente é de interesse explorar suas capacidades para uso com imagens.

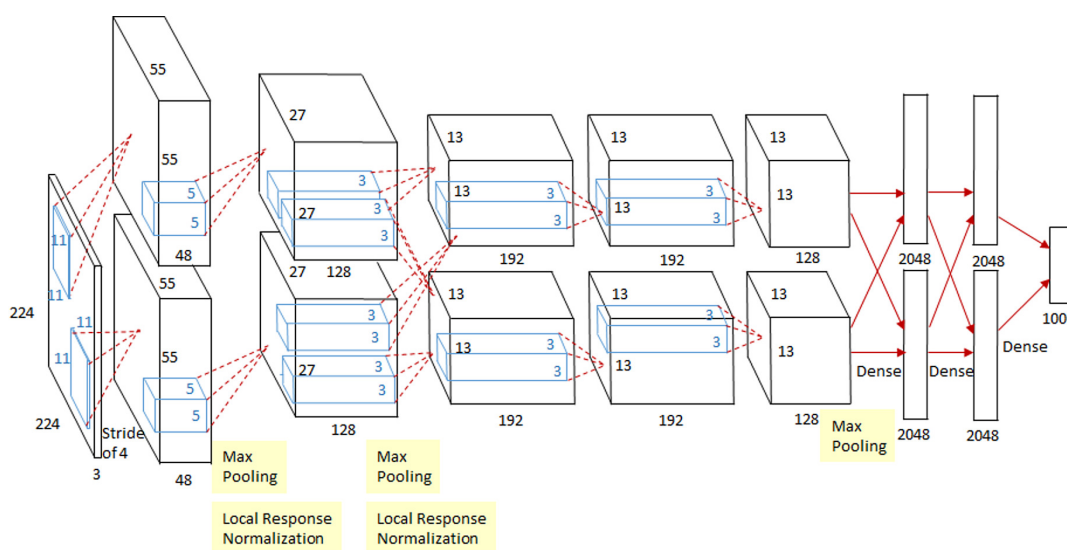
Estruturalmente, uma de suas principais diferenças em relação às redes

² CNN, *Convolutional Neural Network*

do tipo perceptron, é que suas camadas não são completamente conectadas, sendo os neurônios somente conectados de forma local entre si. Para o processamento de imagens isto é uma necessidade, pois o número de conexões entre os neurônios cresce de forma multiplicativa com o tamanho das camadas. Uma imagem de 200x200 requereria aproximadamente 40000 parâmetros *para cada* neurônio da próxima camada. Este agrupamento tem o efeito de que somente píxeis próximos afetam uns aos outros, similarmente ao córtex visual humano. Isto é mais coerente quando se está tentando classificar imagens pois a estrutura hierárquica da rede em si, onde o resultado de uma camada alimenta o próximo, que detecta padrões complexos baseados em classificações simples efetuadas pelas camadas anteriores.

Uma visualização de uma arquitetura clássica de RNC, chamada de `alexnet`, pode ser vista na figura 3. Sua estrutura não convencional, dividida entre 2 caminhos de dados, advém de uma limitação da sua implementação original, pois foi treinada por 6 dias consecutivos em um par de Nvidia Geforce GTX 580 GPUs, razão pela qual a arquitetura de rede original é dividida em duas *pipelines* separadas (WANGENHEIM, 2017).

Figura 3 – Exemplo de uma arquitetura de rede neural convolucional, `alexnet`



Fonte: (WANGENHEIM, 2017)

É possível observar na figura 3 que existe uma divisão entre as 3 últimas camadas, que são densamente conectadas similarmente a um perceptron e o resto da estrutura da rede. Isto acontece pois, ao longo das camadas convolucionais, a matriz (ou imagem) de entrada vai tendo sua dimensão sucessivamente reduzida até se tornar um valor escalar (matriz 1x1). O vetor de saída da ultima camada convolutiva é, então, conectado às camadas densamente conectadas até o valor de saída. Cada um dos valores de saída da parte convolucional representa um filtro que identifica uma característica abstrata que a parte densamente conectada utiliza para determinar o

parâmetro ou classe de interesse.

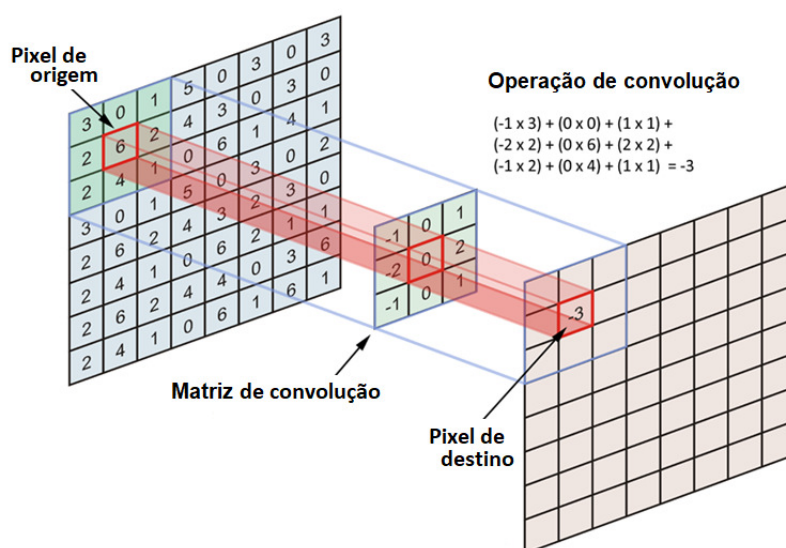
2.1.2.1 Convolução de Kernel

Uma convolução de *Kernel*, é o processo de aplicar uma matriz ou *kernel* sobre uma imagem de modo a obter informações sobre sua estrutura.

A operação em si, é definida pela multiplicação e soma, elemento a elemento de uma pequena matriz sobre uma imagem a ser analisada, se iterando sobre a imagem de entrada até todos os píxeis individuais terem sido analisados, exceto as bordas da imagem. Existem vários métodos para lidar com este problema, mas usualmente esta operação reduz a dimensão da matriz de entrada por $n-1$ em ambos os eixos. Sendo as matrizes de convolução utilizadas sempre de dimensão n ímpar, de modo à possuir um elemento central.

A figura 4 exemplifica a operação para o caso de uma matrix 3x3.

Figura 4 – Exemplo de convolução de kernel

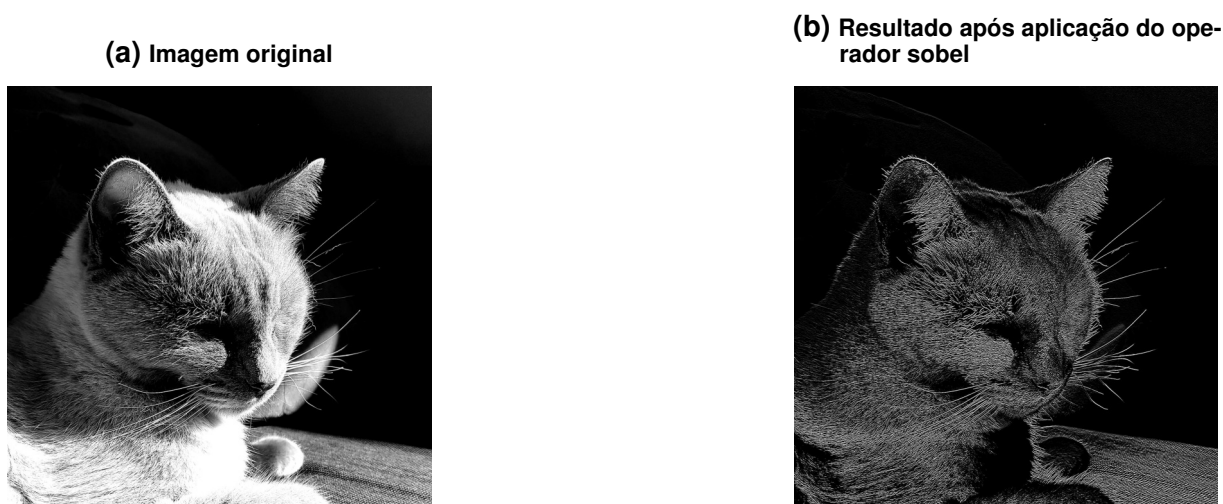


Fonte: Adaptado de (LE MOS, 2021)

Após aplicado um *kernel* sobre uma imagem, o resultado é normalizado por meio de uma função de normalização, detalhada em 2.1.2.3.

Após normalizado, os resultados obtidos são chamados de mapas de características, pois representam a distribuição de uma certa característica abstrata ou padrão intermediário que a rede foi treinada a identificar de forma a classificar um certo elemento de interesse.

Um exemplo clássico, de convolução de *kernel*, é o operador sobel, que resalta todas as mudanças drásticas de cor, ou bordas, em uma imagem, como exemplificado na figura 5.

Figura 5 – Exemplo do operador sobel

Fonte: do autor

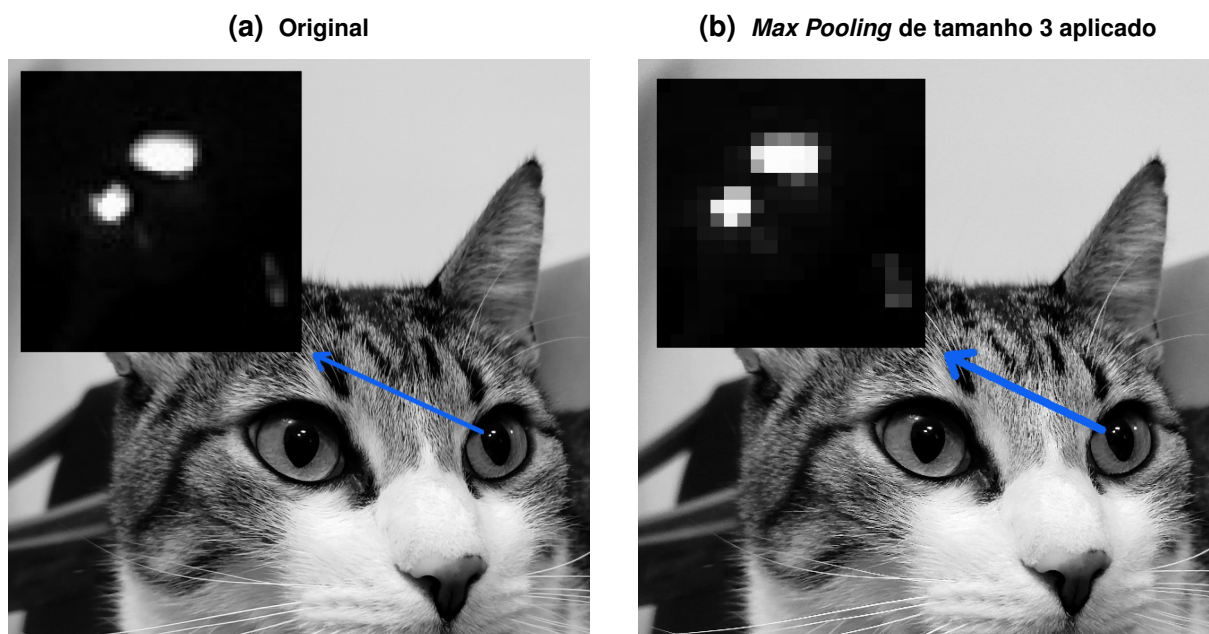
Este *kernel*, ao contrário do processo de treinamento utilizado com redes neurais, foi desenvolvido manualmente e é uma ferramenta muito comum em sistemas de visão clássicos, empregado para auxiliar no processo de segmentação de uma imagem. Inversamente ao operador Sobel, usualmente durante o processo de treinamento de uma RNC, são ajustados iterativamente os valores das matrizes de convolução, que implicitamente se tornam classificadores para características relevantes à aplicação escolhida.

2.1.2.2 *Max pooling*

Max pooling, ou agrupamento máximo, tem o objetivo de reduzir a dimensionalidade das matrizes de saída das camadas convolutivas, comprimindo o mapa de características obtido e reduzindo a demanda de processamento da rede neural. Seu princípio de operação atua de modo a avaliar regiões $n \times n$ da sua matriz de entrada e produzir uma única saída que é simplesmente o maior valor neste bloco. Desta forma, a dimensão da matriz de entrada é efetivamente dividida por n . Pode-se observar nas figuras 6a e 6b o antes e depois da aplicação desta operação.

Também foi incluído no apêndice A um exemplo de como implementar esta operação.

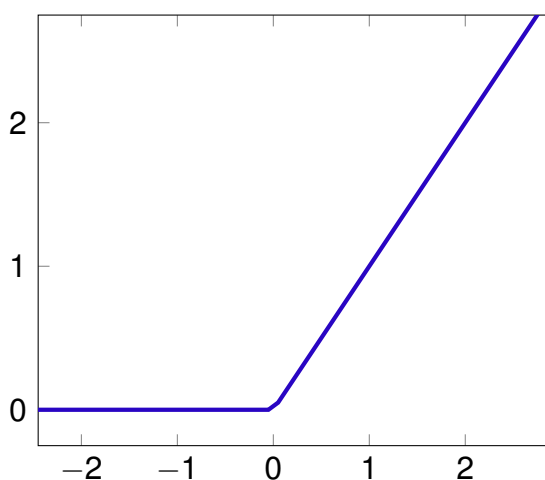
A razão da escolha da função de máximo em vez de qualquer outra operação advém do que estas matrizes representam. Como identificam características da imagem, esta operação sempre seleciona a característica mais fortemente ativada dentro da região avaliada. Isto atua de modo a generalizar o aprendizado da rede e evitar *Overfitting*, que é quando o modelo de aprendizado se especializa demais no conjunto de dados utilizado, perdendo capacidade de generalização.

Figura 6 – Comparação de antes e depois de uma imagem para *Max Pooling*

Fonte: do autor.

2.1.2.3 *Local Response Normalization*

Local Response Normalization é uma operação utilizada em redes neurais com o objetivo de localmente aumentar o contraste do mapa de características, ressaltando estímulos de interesse, reduzindo ativações de baixo valor. Usualmente, esta operação é acompanhada da normalização da matriz de características e, ao contrário dos perceptrons usuais, RNCs em geral usam a função de ativação chamada de ReLu, ilustrado na figura 7, em vez das funções logísticas usuais como $\tanh(x)$.

Figura 7 – Gráfico da função de ativação ReLu

Fonte: do autor

A função é definida por $f(x) = \max(0, x)$, o que efetivamente descarta todos

os valores abaixo de 0. Esta escolha introduz não-linearidades no mapa de decisão da rede, mas sem grandes efeitos negativos. A principal vantagem desta escolha advém do fato que esta função é várias vezes mais rápida de ser computada em comparação com outras funções de ativação sem introduzir uma grande perda de generalidade no modelo resultante.

2.1.2.4 *Uso de recursos*

Principalmente no caso das RNCs, um dos fatores que por muito tempo impediu sua larga aplicação é seu uso de memória e recursos em geral. Mesmo não sendo completamente conectada e otimizando o uso de memória de suas camadas, ainda se demanda uma quantidade considerável de memória para carregar e executar inferências, especialmente se o modelo não for bem otimizado.

Um dos principais aspectos que afetam o uso de recursos de RNCs, principalmente memória, é que de cada camada de convolução da rede produz como saída um grande vetor de matrizes, uma para cada neurônio de entrada. Manter e processar estas matrizes consome uma quantidade significativa de memória e poder de processamento, pois cada operação de convolução demanda uma grande quantidade de operações soma e multiplicação. Estas operações podem ser aceleradas pelo paralelismo de uma GPU e este assunto é explorado no capítulo 2.2.2.

Além disso, é relevante comentar técnicas mais recentes de compressão para redes neurais, uma das mais promissoras, quantização, em que se converte a representação de alta precisão (ponto flutuante de 32 bits) da rede neural em uma representação com menos bits (ponto fixo de 8 bits) (LEE *et al.*, 2022, Adaptado). Este método permite reduzir drasticamente o uso de memória e tempo de execução de aplicações de aprendizado de máquina e em grande parte preservando a sua funcionalidade³.

2.1.2.5 *Transfer Learning*

Existem muitas similaridades e tipos primários de características em comum entre imagens, como detecção de bordas, formas entre outras característica gerais. Por esta razão, se desenvolveu a ferramenta conhecida como *Transfer Learning* ou também “RNAs pré treinadas”, onde grande parte do esforço de treinamento de uma rede já foi feito, e este modelo pré treinado é utilizado como base para alguma especialização ou Transferência para uma tarefa específica, por meio do treinamento de somente suas camadas finais densamente conectadas.

³ Em alguns casos há uma perda de acurácia, mas usualmente é aceitável

2.2 Plataformas computacionais

Atualmente existem diversas plataformas de computação, cada uma com suas características, vantagens e desvantagens. Para o interesse desta avaliação, serão consideradas quatro tipos de plataformas de computação relevantes: CPU, GPU, Placas Aceleradoras e Sistemas Embarcados.

2.2.1 CPU

CPU, *Central Processing Unit* (Unidade de processamento Central), é o elemento de computação mais geral o possível e é o que suporta toda a operação de um sistema. Muitas vezes é o elemento utilizado como opção reserva quando não há outras opções disponíveis, pois por mais ineficiente que seja, é capaz de executar “qualquer tarefa” e todo o sistema a possui.

Atualmente, existem diversas arquiteturas diferentes de CPU como: X86_64, ARM, RISC-V, PowerPC, etc. Cada arquitetura possui suas limitações e especialidades de acordo com seu propósito original.

As duas principais arquiteturas relevantes para este trabalho são a X86_64, que é utilizada com quase todos os computadores modernos e é focada em performance e uso geral, e a ARM, que é mais comum em aplicações focadas em eficiência energética e simplicidade⁴, como celulares e dispositivos embarcados.

2.2.2 GPU

GPU, *Graphics Processing Unit* (Unidade de Processamento Gráfico), é um dispositivo de computação especializado em tarefas relacionadas com gráficos 2D/3D. Quando integrado em um sistema, em grande parte depende de outros dispositivos, em particular a CPU para efetuar qualquer tarefa de interesse e não possui grande poder de computação geral.

Por virtude da sua estrutura ser focada em pequenas tarefas executadas em paralelo, pode ser utilizado para acelerar tarefas relacionadas ao aprendizado de máquina. Especialmente no caso de RNC's, esta relação é altamente benéfica, pois este tipo de rede utiliza operações comuns ao processamento de imagens que esta plataforma se especializa.

Atualmente é uma das melhores plataformas existentes para o treinamento de redes neurais, por ser largamente disponível e bem adaptada à esta tarefa. Também é relevante comentar que várias GPUs mais recentes⁵ possuem *hardware* de propósito específico para acelerar tarefas de aprendizado de máquina e similares.

⁴ Existem diversas versões e variações desta arquitetura com capacidades diferentes, mas o foco é similar.

⁵ Especialmente da fabricante NVIDIA.

2.2.3 Placas Aceleradoras

Além dos elementos clássicos presentes em um computador, cada vez mais vem surgindo novos dispositivos e placas aceleradoras especificamente pensados para *machine learning*. Alguns sendo capazes de treinamento e inferência, como a série A da Nvidia (mais comuns no espaço do data center), outros somente para inferência como a plataforma Google Coral⁶, entre outros.

2.2.4 Sistemas Embarcados

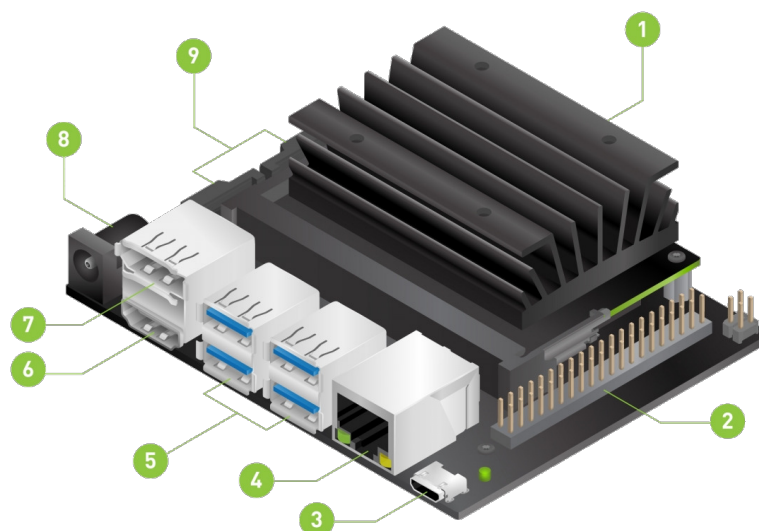
Por último, também existem sistemas embarcados inteiros especificamente pensados para aplicações aprendizado de máquina, que podem ser utilizados para computação local. Possuem pequenos chips de baixo consumo energético e propósito específico, capazes de realizar eficientemente inferências de modelos já treinados. Este trabalho explorou primariamente as capacidades de uma das placas da plataforma embarcada híbrida da Nvidia, chamada de NVIDIA Jetson NANO como sistema embarcado de exemplo.

Esta placa traz *hardware* de uma GPU da família Maxwell⁷, integrada em um sistema embarcado com uma capacidade de memória respeitável de 4Gb e uma CPU *arm64*. Além disso, todo o sistema está limitada em torno de 15W de potência total disponível. Sua folha de dados sumarizada está disponível no anexo A.

Pode-se ver na figura 8 um diagrama apresentando suas interfaces externas.

⁶ Que também oferece os seus chips destinados à inferência para integração em outros sistemas

⁷ Geforce GTX 9xx

Figura 8 – Diagrama da placa de desenvolvimento NVIDIA Jetson NANO.

Fonte: Website oficial do fabricante, Acesso em 30/06/2022.

Sendo suas descrições:

- | | |
|---|--|
| 1 - slot μ SD para armazenamento principal | 5 - Portas USB 3.0 x 4 |
| 2 - header de 40 pinos para expansão | 6 - Porta HDMI de saída |
| 3 - porta μ USB para alimentação 5V ($\leq 5W$) ou modo dispositivo | 7 - Porta DisplayPort |
| 4 - porta <i>Ethernet Gigabit</i> | 8 - Conector P4 para alimentação 5V ($\leq 15W$) |
| | 9 - Conector MIPI CSI-2 para Câmeras |

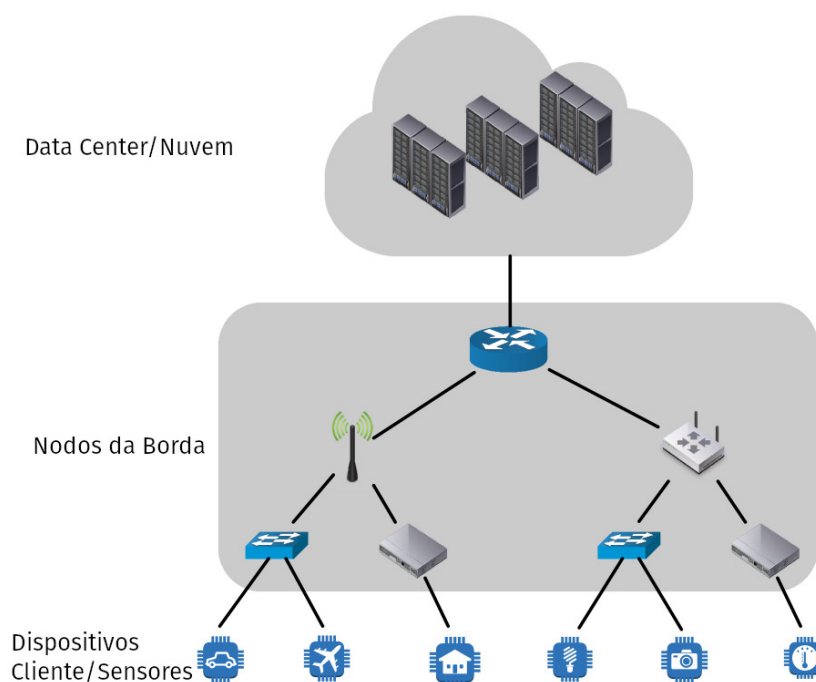
Sua arquitetura utiliza um banco de memória compartilhado entre a GPU e CPU, contrário à maior parte dos computadores, onde possuem bancos próprios isolados. Isto permite melhor eficiência no compartilhamento deste recurso limitado entre os elementos de computação.

Além disso, também é relevante comentar que existem outros modelos na família Jetson (como a NVIDIA Jetson Xavier), com *hardware* de capacidade melhor, com mais memória, melhor performance e arquitetura mais recente da GPU, mas não foi financeiramente possível explorar estas opções no presente momento.

2.3 Computação em borda e aplicações de visão computacional

Computação em borda é uma metodologia para estruturação de sistemas onde se leva as aplicações de processamento de dados para mais perto das suas fontes (chamada de borda) e consumidores (como celulares, dispositivos IOT e máquinas industriais), por meio de pequenas unidades de processamento e servidores localizados na borda, como exemplificado na figura 9

Figura 9 – Arquitetura de um sistema com processamento de borda



Fonte: Adaptado de (HONG; VARGHESE, 2019)

Este paradigma surgiu para tentar atender os desafios trazidos pela quantidade de novos dispositivos IoT (*Internet of Things*) e fontes de dados em geral, sendo adicionados à infraestrutura moderna, como em ambientes comerciais, residenciais, industriais, etc e como atendê-los e integrá-los entre si. O paradigma anterior, de se depender de uma “nuvem” externa estava se provando limitado frente aos novos desafios propostos.

Deste modo, procurou-se tentar distribuir e trazer esta tarefa de processamento para mais perto de onde os dados são produzidos, na ‘borda’.

Os principais diferenciais de se utilizar este paradigma, são:

- Privacidade dos dados processados: muitas vezes, as informações a serem processadas podem ser consideradas sensíveis, por questões legais (como dados de imagem de pessoas) ou por questões de sigilo industrial. O processamento

local remove a necessidade de transportar e expor estes dados ao risco de vazamento.

- Volume de dados: Mesmo com o crescimento da disponibilidade do acesso à rede na borda, muitos sensores e tipos de demanda necessitariam a transmissão de um volume desproporcional de dados através da rede para obtenção de um resultado significativo, pois a densidade de informações relevantes em um dado sensor é baixa.
- Latência: muitos processos são sensíveis à latência, e quanto menor o atraso entre produção do dado e resultado da análise, melhor a eficiência do mesmo.
- Robustez: depende-se menos de infraestrutura externa para o funcionamento de um processo/operação.

Tendo em vista as vantagens supracitadas, um dos principais desafios do processamento de borda, atualmente, ainda vem da heterogeneidade dos sistemas envolvidos. Como cada vez mais máquinas e equipamentos industriais, comerciais e residenciais têm sido projetados com capacidade de processamento integrada substancial, se abriu uma porta para se utilizar deste poder de processamento para estes dispositivos atuarem também como servidores *edge*. Mas como este tipo de sistema varia muito em capacidades, arquitetura e sistema operacional, entre outros, é difícil implementar aplicações de forma escalável, sem demandar muita configuração e adaptação da aplicação de interesse para cada equipamento individual⁸.

Considerando esta demanda, ao longo do tempo vêm se desenvolvendo sistemas capazes de auxiliar a execução destas tarefas de forma mais escalável e consistente, como servidores de borda dedicados. Inicialmente baseados em arquiteturas conhecidas, como PC, Mini-PC⁹, mas também cada vez mais especializados. Desta forma que possam ser mais eficientes, potentes (principalmente devido ao avanço da tecnologia) e direcionados para sua tarefa específica (como aceleradores de IA), se tornando capazes de executar algoritmos mais complexos e relevantes às demandas reais, além de também abrir novas possibilidades antes não consideradas devido a limitações técnicas.

Um exemplo de possível demanda de borda, seria a proposta descrita no artigo (STIVANELLO; MASSON; STEMMER, 2022), onde é implementado um sistema de detecção de falhas baseado em RNC que o sistema de processamento é acessado via rede. Caso seja possível atingir o tempo de inferência demandado, neste caso de 4 à 8 Hz¹⁰, com um sistema embarcado, seria um excelente exemplo do tipo de demanda

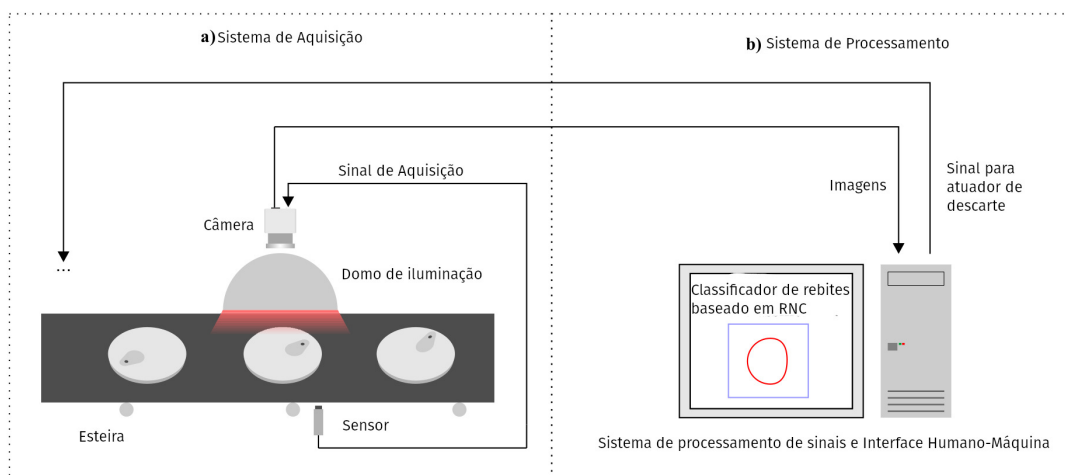
⁸ Mas existe estudos e esforço no sentido de facilitar isso, como a plataforma da IBM

⁹ como Intel NUC

¹⁰ 0.250 s à 0.125 s.

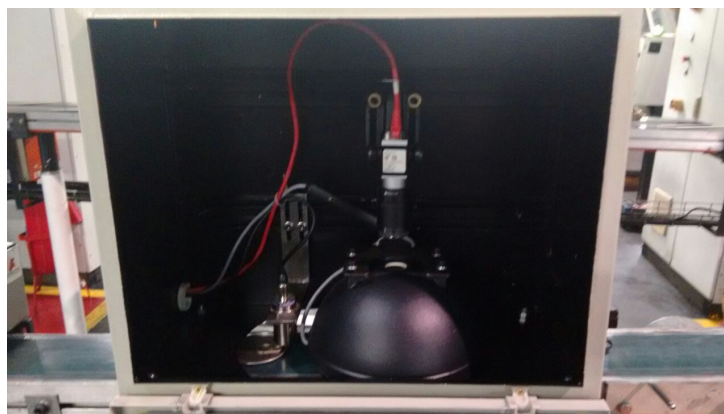
que o processamento de borda pode suprir. A arquitetura originalmente proposta pode ser vista na figura 10, onde o sistema de processamento remoto seria substituído por um sistema embarcado ao lado da própria esteira. Pode-se ver também na figura 11 a implementação física do sistema.

Figura 10 – Arquitetura do sistema proposto no Artigo



Fonte: Adaptado de (STIVANELLO; MASSON; STEMMER, 2022).

Figura 11 – Implementação física do sistema proposto no Artigo



Fonte: Adaptado de (STIVANELLO; MASSON; STEMMER, 2022).

Dado este exemplo, hoje em dia, a tendência tem sido de crescimento no uso e na implantação de sistemas de aprendizado de máquina, pois os requisitos funcionais impostos pelas demandas tem inviabilizado ou tornado obsoleto o uso de visão clássica.

Esta nova metodologia, de forma simplificada, somente tem como demanda primária a classificação manual dos dados para serem utilizados no treinamento de um modelo e a escolha da estrutura do modelo, o que facilita a implementação de novos sistemas por ser mais generalizável.

Significativamente, uma mudança recente tem sido que sistemas de borda (especializados ou não) tem viabilizado a implantação destas ferramentas em ambientes comerciais e industriais sem depender tanto de infraestrutura remota.

2.4 Trabalhos correlatos

Oportunamente, vários trabalhos recentes vêm tratando de temáticas relacionadas ao uso de RNCs, explorando suas diversas aplicações possíveis.

Um primeiro exemplo recente seria o desenvolvido por Ramon Packer (BRIGNOLI, 2021), que teve como objetivo desenvolver um classificador de imagens automático para tipos de ambientes de um imóvel, como quarto, sala, cozinha, banheiro fachada e exterior, com a intenção de auxiliar no processo de digitalização das imobiliárias. Ele conseguiu atingir uma precisão de classificação de 96.94% e tempo médio de processamento de somente 80 ms utilizando uma RNC de 50 camadas, treinada em 10 *epochs* com um banco de dados de quase 16.000 imagens divididas entre 6 categorias.

Um outro trabalho proeminente, seria de Yessica Valencia (LEMOS, 2021), que se propõe a comparar diversas técnicas para identificação de defeitos e controle de qualidade em ovos, principalmente avaliando sistemas de visão clássicos em comparação com RNCs e qual atinge um melhor balanço entre complexidade, usabilidade, precisão e tempo de execução. Obteve resultados de precisão média de detecção de até 91.7 % para detecção de sujeiras, 91.9 % na detecção de anomalias de formato e 94.5 % na detecção de fissuras. Concluiu finalmente que, em geral, o uso de RNCs é mais simples de implementar, pois demanda somente imagens classificadas, mas nem sempre atende os requisitos de tempo, que no caso era de 60Hz¹¹.

Também é relevante ressaltar o trabalho de Jinsoo Kim e Jeongho Cho (KIM; CHO, 2021), que implementaram um sistema de detecção de comportamento humano baseado na análise de vídeo por meio de RNCs. Este sistema foi pensado para ser utilizado dentro de sistemas embarcados, o que permite serem posicionados em conjunto da câmera, e efetuar esta análise em tempo real. Devido a demandas decorrentes do uso em ambiente embarcado, se explora uma arquitetura que extrai *features* espaço-temporais de acordo com diferenças entre os *frames* de vídeo e isto, depois, alimenta uma rede convolucional com o resultado. Este artigo utilizou o mesmo sistema embarcado que o presente trabalho pretende explorar, uma NVIDIA Jetson Nano, para representar as demandas resultantes de se utilizar um ambiente embarcado. Mesmo com estas restrições, os tempos de detecção para ações são razoáveis, todos abaixo de 2s.

¹¹ Tempo máximo de execução de 16.6 ms

E, finalizando, são encontrados, na literatura, muitos outros trabalhos que tratam de sistemas de visão implantados em sistemas embarcados. Como exemplo podemos citar o trabalho (RAMOS; STIVANELLO; STEMMER, 2020), onde a aplicação refere-se à detecção automática de defeitos em rebites. Tal sistema foi desenvolvido em FPGA, entretanto sem utilizar algoritmos de aprendizado de máquina. Uma nova versão do trabalho, (STIVANELLO; MASSON; STEMMER, 2022), foi desenvolvida uma solução empregando RNC com o mesmo objetivo, porém sem que tenha sido embarcado, como exemplificado anteriormente em 2.3. Seu banco de dados será utilizado como uma das aplicações a serem avaliadas.

3 METODOLOGIA

Neste capítulo é descrita a metodologia de desenvolvimento e todo o processo envolvido na implantação de um sistema de aprendizado de máquina baseado em RNC em ambiente embarcado.

Primeiramente serão explorados quais critérios foram considerados para comparação entre as plataformas. Em seguida, na seção 3.2, será descrito o processo de configuração e adaptação do ambiente de desenvolvimento de RNC escolhido, para o uso em ambientes embarcados e os desafios envolvidos. Subsequentemente, na seção 3.3, serão abordados e explorados brevemente os bancos de dados escolhidos para esta implementação e suas características. Logo após, na seção 3.4, será documentado o processo de configuração do ambiente de avaliação utilizado na plataforma escolhida. Por último, na seção 3.5, serão detalhados os resultados experimentais obtidos.

3.1 Caracterização dos critérios a serem avaliados nos testes

De modo a avaliar as plataformas escolhidas, foram definidos alguns critérios para uso como ferramenta de comparação. Os critérios selecionados são apresentados nas seções a seguir, bem como suas justificativas e relevância.

3.1.1 Potência/Consumo de energia

A relação Potência/Consumo ainda é uma das principais características que diferenciam sistemas embarcados de computadores usuais. Em geral, seu consumo e oferta máxima de potência disponível, são reduzidos em comparação à plataforma PC.

Além disso, é um fator extremamente relevante para aplicações móveis e/ou dependentes de baterias e similares, e mesmo quando não há restrições diretas ainda é fortemente relacionada à seção 3.1.4, pois em quase todos os casos a demanda de ventilação é diretamente proporcional à potência dos elementos de processamento.

Dado o mesmo silício e capacidades, muitas vezes o que efetivamente determina o teto de performance de um sistema é este parâmetro.

3.1.2 Tempo de execução

Em muitas aplicações, o tempo de execução é um dos principais indicadores de viabilidade de uma dada solução, especialmente em demandas onde é necessário processamento de dados em tempo real. Mesmo em casos onde não existe este requisito estrito, este fator sempre possui algum grau de relevância.

3.1.3 Uso de memória

Como sistemas embarcados em geral possuem recursos reduzidos e de mais difícil expansão¹, em comparação com PC, é crítico que a aplicação seja pensada para não exceder a capacidade de memória disponível no sistema a ser utilizado. Além disso, é um requisito de difícil adaptação, pois é um recurso muitas vezes intrínseco ao sistema e de complexa substituição. Foram brevemente comentados na seção 2.1.2.4 possíveis métodos para se tentar reduzir a demanda de memória de uma dada aplicação.

3.1.4 Ventilação

Como observado anteriormente em 3.1.1, a demanda por ventilação de um sistema é quase sempre² diretamente proporcional à sua potência.

Este parâmetro de comparação é particularmente relevante em aplicações externas (ambientes externos), industriais e de atmosfera insalubre, em geral. A disponibilidade de ventilação em geral é reduzida para evitar o ingresso de umidade e detritos nas partes eletrônicas sensíveis de um sistema. Também existe, em alguns casos requisito de não existência de faíscas, que pode ser causado por estática, movimentação de ar e motores, como em fábricas onde se lida com produtos inflamáveis³.

3.1.5 Dimensão física

Devido ao avanço da tecnologia e disponibilidade de sistemas compactos, o volume físico não é mais um fator tão decisivo para seleção entre PC e plataformas embarcadas. Existem sistemas ainda considerados 'PCs', de tamanho diminuto, mas, usualmente, sistemas embarcados ainda tendem a ocupar um volume menor em comparação. Existem aplicações onde este pode ser um fator determinante na escolha de plataforma de processamento.

3.1.6 Custo Financeiro

É uma métrica relevante, pois, limita a aplicabilidade de uma dada plataforma em relação ao seu custo/benefício. Por exemplo, uma plataforma pode consumir uma pequena quantidade de energia, possuir uma grande quantidade de memória, ser compacta e ter bons tempos de execução, mas, não ser viável devido ao alto custo financeiro associado para uma demanda de baixo valor agregado.

¹ Pois muitas vezes são sistemas integrados, não modulares

² Existem soluções totalmente passivas (sem ventilação e partes móveis), mas em geral são incomuns por serem caras e de grandes dimensões físicas

³ e.g.: Fábricas de tinta

3.2 Ferramentas Utilizadas

Esta seção detalha as ferramentas e processos utilizados, documentando sua seleção e configuração além de outros ajustes técnicos implementados. Será descrito o processo de configuração da plataforma escolhida, seguido das aplicações escolhidas para avaliação.

Visando poder rapidamente desenvolver, treinar e avaliar modelos de aprendizado de máquina, foi utilizada a linguagem de programação `python` e o *framework* de aprendizado de máquina `FastAI`, que é “uma biblioteca de aprendizado profundo provendo seus usuários com componentes de alto nível que podem rápida e facilmente prover resultados de ponta em domínios padrão do aprendizado de máquina, além de prover pesquisadores com componentes de baixo nível que podem ser combinados para construir novas abordagens” (HOWARD; GUGGER, 2020, Adaptado)

3.2.1 Plataformas Avaliadas

Com o intuito de comparar arquiteturas clássicas de computação com tecnologias embarcadas, foram estudadas plataformas de inferência presentemente disponíveis no mercado e selecionadas duas, de acordo com disponibilidade, aplicabilidade e recursos existentes: as plataformas PC de propósito geral⁴ e a placa NVIDIA Jetson NANO. Para representar plataformas embarcadas, foi selecionado o kit de desenvolvimento NVIDIA Jetson NANO, que tem suas especificações descritas no anexo A, pois possui um bom misto entre sistema embarcado, capacidade de processamento dedicada e recursos em geral, além de ser “largamente disponível no mercado”. Também foi levado em conta a plataforma Google Coral, mas não foi possível a obtenção de uma unidade para avaliação.

No caso do PC utilizado para treinamento e comparação, o mesmo tem suas especificações descritas na tabela 1.

Tabela 1 – Especificações do computador utilizado para comparação

Parâmetro	Valor
OS @ Kernel	Pop!_OS 21.10 x86_64 @ 5.17.5-76051705-generic
CPU	AMD Ryzen © 3 2200G (4 cores, 4 threads) @ 3.500GHz
GPU	MSI NVIDIA GeForce GTX 1050 Ti (4GB VRAM)
RAM	15994MiB (16 GB)

Também é relevante comentar que, inicialmente, foi considerado a inclusão de um *Raspberry PI 3 Model B*, de modo a comparar com uma plataforma largamente utilizada e conhecida no ambiente dos sistemas embarcados, mas devido a demandas

⁴ Ambos os casos de somente CPU ou CPU + GPU

de memória e processamento do tipo de aplicação escolhido, não foi viável a inclusão desta plataforma na comparação. Além disso, apesar de possuir `python` atualizado, encontra problemas de suporte nativo de *software* similares aos descritos em 3.4. Uma sugestão para um trabalho futuro seria tentar utilizar o *Raspberry PI 4 Model B*, que possui mais recursos o modelo 3, mas não foi uma opção financeiramente viável no momento do desenvolvimento do trabalho.

3.2.2 Processo de Inferência e Treinamento

Para o propósito deste trabalho, somente é relevante a inferência de RNCs em plataformas embarcadas, pois é a parte que efetivamente é utilizada em uma aplicação final. Em geral, todo o processo de treinamento das RNCs foi efetuado em ambientes não-embarcados, devido a grande demanda de poder computacional imposta pela realização deste passo.

Também é relevante comentar que, de modo a avaliar as aplicações escolhidas, serão utilizado as arquiteturas de RNC chamadas de `resnet`, Respecificamente a `resnet18`, `resnet32` e `resnet50`, pois originalmente já eram a opção escolhida ao elaborar estes bancos de dados. A descrição do seu princípio de funcionamento e estrutura interna são tópicos fora do escopo desta avaliação.

3.3 Aplicações a serem avaliadas

Nesta seção, são detalhadas quais aplicações/bancos de dados foram escolhidas para comparar a performance entre plataformas e suas justificativas de escolha.

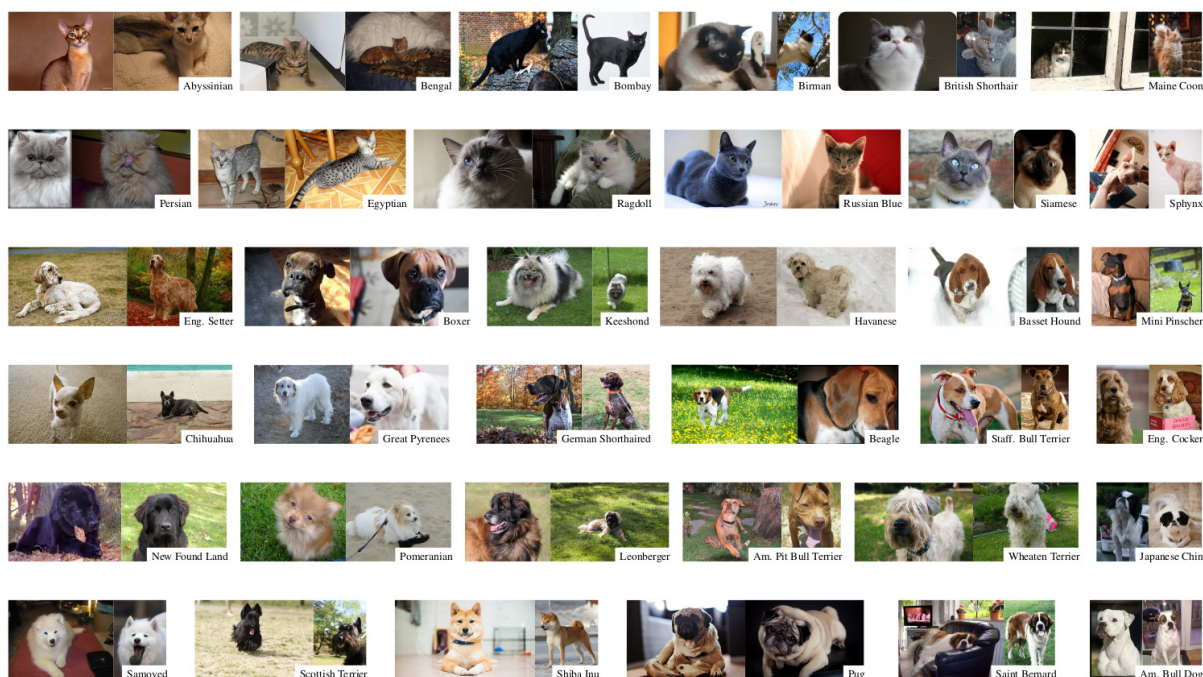
3.3.1 Oxford-IIIT Pet

Este banco de dados foi elaborado “de forma a estudar discriminação de diferentes raças de cachorros e gatos, um exemplo desafiador de categorização fina de objetos” (PARKHI; VEDALDI; JAWAHAR, 2012, Adaptado). Possui em suas classes 25 raças de cachorros e 12 raças de gatos, detalhado na tabela 2, com aproximadamente 200 imagens para cada categoria individual. Pode-se ver na figura 12 uma amostra extraída do banco de dados.

O método de rotulação escolhido para identificar se uma foto é de um gato ou se é de um cachorro foi que, se a primeira letra do nome do arquivo é capitalizada, é a imagem de um gato, se não, é de um cachorro. Ex: `Egyptian_Mau_145.jpg` ou `american_bulldog_105.jpg`

Além disso, Para execução deste trabalho, este banco de dados somente será utilizado para gerar um classificador binário gato x cachorro, devido às restrições de tempo e RAM envolvidos no treinamento e execução do classificador completo.

Figura 12 – Amostra de imagens do banco de dados *Oxford-IIIT Pet*, duas para cada categoria disponível



Fonte: Adaptado de (PARKHI; VEDALDI; JAWAHAR, 2012).

3.3.2 Inspeção de rebites

Este banco de dados foi elaborado de modo a estudar a implementação de um sistema para detectar defeitos de fabricação em rebites de tampas de latas para alimentos utilizando RNCs. Ele contém imagens de rebites que estão ou não com um dado defeito de fabricação, chamado de fratura. Na base de imagens, o rebite já está centralizado na região de interesse (*Region of Interest*, ROI) da tampa, como se pode verificar nas figuras 13b e 13c. Além disso, também é apresentada a tampa sem o ROI na figura 13a. O processo de detecção e localização da ROI foi executado por meio de um algoritmo de visão computacional clássica, chamada de análise de *blob* (STIVANELLO; MASSON; STEMMER, 2022).

Como as imagens já estão centradas somente no ROI do rebite, são todas relativamente pequenas, de dimensão 44 x 44 píxeis. Isto reduz bastante a demanda de recursos da rede neural, acelerando significativamente o treinamento e reduzindo o tamanho e tempo de execução do modelo final.

O método de discriminação das imagens entre fraturadas e não fraturadas, é feito por meio de uma palavra no início de seu nome, *frat* ou *nfrat*. Ex: *nfrat_254Image__2017-07-27__10-23-35roi_rot_180.bmp* e *frat_96Image__2017-07-27__10-58-58roi.bmp*.

Além disso é relevante comentar que, para cada imagem única no banco

Tabela 2 – Composição banco de dados Oxford-IIIT Pet

Gatos		Cachorros	
Raça	Contagem	Raça	Contagem
Abyssinian	198	English Setter	200
Bengal	200	German Shorthaired	200
Birman	200	Great Pyrenees	200
Bombay	184	Havanese	200
British Shorthair	200	Japanese Chin	200
Egyptian Mau	190	Keeshond	199
Maine Coon	200	Leonberger	200
Persian	200	Miniature Pinscher	200
Ragdoll	200	Newfoundland	196
Russian Blue	200	Pomeranian	200
Siamese	199	Pug	200
Sphynx	200	Saint Bernard	200
Soma	2371	Samoyed	200
		Scottish Terrier	199
		Shiba Inu	200
		Staffordshire Bull Terrier	189
		Wheaten Terrier	200
		Yorkshire Terrier	200
		Soma	4978

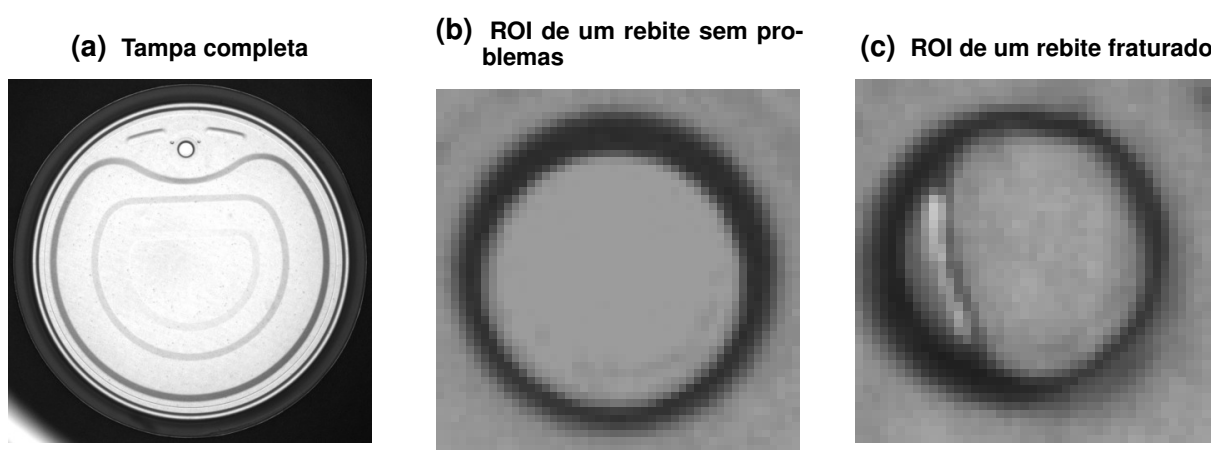
Fonte: Adaptado de (PARKHI; VEDALDI; JAWAHAR, 2012)

de dados, existem 5 cópias suas: rotacionada em 3 possíveis ângulos e espelhada horizontalmente e verticalmente. Isto é feito de modo a aumentar o tamanho do banco de dados, de forma que a rede neural melhor generalize a sua categorização, reduzindo a demanda por dados reais. Esta ferramenta é muito útil, pois modelos de aprendizado de máquina dependem de grandes volumes de dados para evitar o fenômeno chamado sobreajuste (*Overfitting*), onde o modelo se especializa demasiadamente no seu conjunto de dados de treinamento e perde capacidade de generalização, mas muitas vezes a disponibilidade de dados rotulados é limitada (YANG *et al.*, 2022, Adaptado)

A contagem total de imagens originais, não rotacionadas ou espelhadas, e a divisão entre as categorias está descrito na tabela 3.

3.4 Processo de configuração do ambiente

Nesta seção é descrito o processo de configuração das ferramentas do ambiente para cada arquitetura utilizada.

Figura 13 – Exemplos de imagens do banco de dados e da tampa sem o ROI

Fonte: Adaptado de (STIVANELLO; MASSON; STEMMER, 2022)

Tabela 3 – Distribuição dos dados

Tipo	Contagem
Fraturado	47
Não Fraturado	111
Total base	158
Total com variações	949

Fonte: Banco de dados (STIVANELLO; MASSON; STEMMER, 2022)

3.4.1 NVIDIA Jetson NANO

3.4.1.1 Geral

A configuração inicial do sistema operacional da placa NVIDIA Jetson NANO é muito similar a maior parte das distribuições Linux atualmente existentes, pois é baseada no Ubuntu 18, e é, em geral, simples e funcional.

Como a placa de desenvolvimento a ser avaliada é um sistema embarcado e a aplicação não demanda interface gráfica (GUI, *Graphical User Interface*), após feita a configuração inicial, alguns serviços não utilizados como a GUI foram desabilitados para economizar RAM e processamento, além de remover diversos pacotes não-necessários visando economizar espaço em disco. O acesso à placa é feito por meio de conexões SSH (*Secure Shell*) em modo texto e Jupyter Notebooks via rede.

Uma última observação seria que, como é utilizada a biblioteca `python matplotlib` para mostrar gráficos como amostras do conjunto de treinamento e resultados, foi utilizada uma pequena função para salvar os mesmos como imagens em disco, que podem ser visualizadas para obter o resultado. Este ajuste foi documentada aqui: (DEBUGGING TISSUE, 2020).

```
1 import matplotlib.pyplot as plt
```

```
2
3 # ...
4
5
6 # chamar função que geraria um gráfico, como dls.show_batch()
7 # antes e executar logo em seguida:
8 plt.savefig('nome-do-arquivo.png')
9
10 # ...
```

No caso do acesso via Jupyter Notebooks isto não foi necessário.

3.4.1.2 Ferramentas de AI

De modo a instalar e configurar o ambiente de desenvolvimento para inteligência artificial nesta plataforma, foram necessários vários ajustes e adaptações técnicas ao procedimento oficial disponibilizado em (FAST.AI, 2022), que serão descritos na subseção a seguir.

O principal entrave neste processo adveio do fato de que a fabricante da placa, NVIDIA, somente fornece uma versão relativamente defasada da linguagem python. Esta versão, 3.6.9 de 2 Julho de 2019, portanto há mais de 3 anos, já é considerada EOL (*End Of Life*, Fim de Vida) pelo desenvolvedor da linguagem de programação.

Como para a reduzir o esforço de manutenção, versões mais recentes de bibliotecas tendem a atualizar junto da versão mais recente da linguagem e não oferecerem suporte às versões antigas do interpretador. Neste caso, várias das bibliotecas utilizadas, incluindo FastAI, não são compatíveis em sua versão mais recente disponível.

Isto não seria problemático, se não fosse o fato de que atualmente muitas vezes não é documentada ou checada a compatibilidade de uma dada versão de uma biblioteca com uma dada versão da linguagem durante seu processo de instalação. Um bom exemplo deste problema pode ser verificado na principal biblioteca utilizada, FastAI, que não possui nenhuma checagem de versão durante o seu processo de instalação. Em um dado caso o *update* da versão 2.5.4 para a 2.5.4, ou seja, somente aumentando o identificador de correção (WERNER, 2018), introduziu uma mudança que, silenciosamente, incompatibiliza utilização da biblioteca com versões do interpretador anteriores à 3.7, documentado neste *ticket* de suporte aberto no site que hospeda o código fonte da biblioteca.

Inicialmente foi considerado instalar uma versão mais recente do interpretador, por meio de um ambiente criado pela ferramenta Anaconda, ou por meio de um PPA (*Personal Package Archive*) do gerenciador de pacotes APT, para resolver os problemas de compatibilidade. Mas, em ambos os casos, foi encontrado um impedimento: uma

das bibliotecas que a FastAI depende, PyTorch, é essencial que esteja compilada em sua versão com suporte para GPU, caso contrário o *hardware* especializado da placa não é utilizado e tudo é feito na CPU, o que não é desejável nem eficaz. Como a NVIDIA, fabricante da placa, somente fornece versões compatíveis com python até 3.6, seria necessário ou utilizar a versão defasada e tentar “descobrir” por tentativa e erro qual versão da FastAI é compatível com este interpretador, ou tentar manualmente compilar a biblioteca, o que seria complexo e particularmente dispendioso se feito no processador do sistema embarcado ou ainda mais complexo no caso de se tentar utilizar compilação cruzada em um sistema de arquitetura diferente, com suporte à GPU para uma versão mais recente do interpretador⁵.

Após muitas tentativas e erros, a solução encontrada foi utilizar um ambiente Anaconda python 3.6.9⁶, com FastAI 2.5.5⁷. De modo a obter suporte à GPU, foi primeiro instalado FastAI e suas dependências normalmente, e posteriormente, reinstalada a versão correta da biblioteca PyTorch baixada do site da NVIDIA seguindo o artigo oficial sobre a instalação desta biblioteca. Também é relevante comentar que, como algumas outras bibliotecas precisam ser compiladas para a arquitetura-alvo a ser utilizada - como, por exemplo, spacy e cymem - durante a instalação houve o *download* e a compilação das mesmas, o que aumenta significativamente o tempo de instalação. Além disso, em alguns casos, a compilação falhou de forma críptica⁸. Devido a diferença de versão, alguns dos elementos e funções usados nos guias e na documentação mais recente da FastAI possuem diferenças de nomenclatura e/ou sintaxe, um exemplo principal é a função que presentemente se chama *vision_learner*, utilizada para criar modelos de aprendizagem mas na versão utilizada chama-se *cnn_learner* e suporta os mesmos parâmetros do novo nome ou a classe utilitária *DataLoader*, previamente *DataBunch*, que mudou significativamente de uma versão para outra incluindo nomenclatura, sintaxe e parâmetros utilizados.

Também foi tentado compilar o binário .whl do PyTorch para uma versão mais recente do python, mas mesmo depois de um processo de 12h de compilação, o resultado obtido não foi bem sucedido.

3.4.2 PC

Na plataforma PC, a instalação do ambiente de trabalho para inteligência artificial foi um processo simples, principalmente porque todo o *software* utilizado é recente, sendo suficiente seguir o procedimento ofertado no site do desenvolvedor

⁵ Durante o desenvolvimento do projeto ocorreu uma situação onde houve uma instalação completa bem sucedida, mas a versão do PyTorch instalada não possuía suporte à GPU

⁶ Atualmente na versão 3.10.5

⁷ Atualmente na versão 2.7.5

⁸ Não de fácil resolução

da ferramenta (FAST.AI, 2022) sem grandes problemas. O gerenciador de pacotes utilizado foi o `conda` (CONDA, 2022).

3.5 Resultados experimentais

Nesta seção, primeiramente, serão apresentados os resultados obtidos no treinamento e na inferência sobre os bancos de dados delineados anteriormente em 3.3. Em seguida será discutida a relação entre os requisitos levantados anteriormente e os valores obtidos.

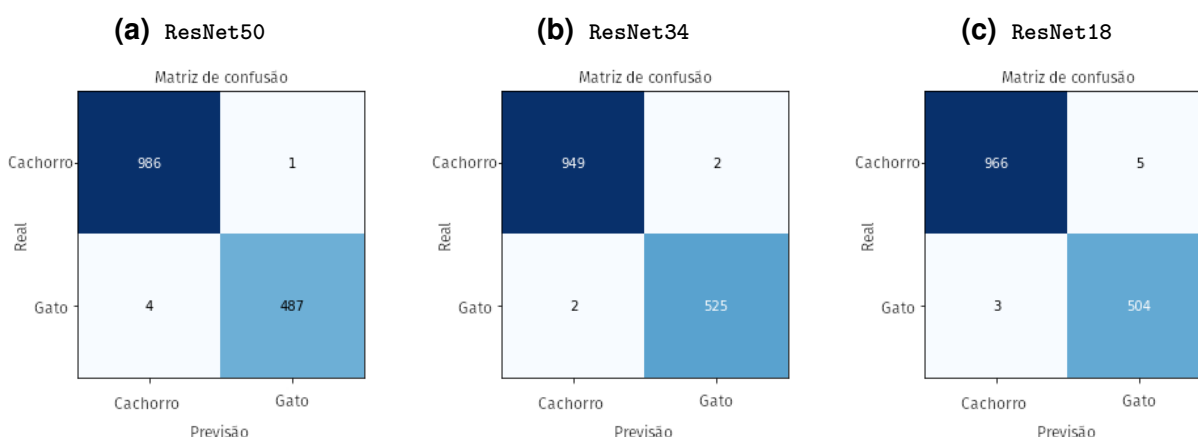
Primeiramente, serão apresentados os resultado do treinamento: gráficos de perda e matriz de confusão para cada modelo de RNC avaliado e em seguida detalhado, numa tabela, os resultados de tempo de inferência, precisão e uso de memória.

3.5.1 Oxford-IIIT Pet

Devido a grande quantidade de imagens, este modelo demandou mais memória disponível durante o treinamento. Além disso, o tamanho de *batch* teve que ser ajustado, para ser possível o seu treinamento utilizando `resnet50`.

Em ambos os casos, a validação do tempo de inferência foi feito por meio de 10 rodadas de 100 iterações, para obter um valor médio.

Figura 14 – Matrizes de confusão para cada arquitetura de RNC avaliada



Fonte: Elaborado pelo autor

Pode-se observar na tabela 4 que utilizar a arquitetura de profundidade média, `resnet34`, é o ideal, pois na arquitetura seguinte a precisão de categorização cai. Usualmente, quando se compara dois modelos similares de profundidades diferentes, quanto maior a quantidade de camadas da arquitetura escolhida maior seu tempo de inferência e treinamento, além de demanda por memória de trabalho.

Além disso, pode-se observar no anexo C, o código utilizado no processo de treinamento da rede neural. Também foi elaborada uma versão somente para inferência,

Tabela 4 – Tempos de execução - Oxford-IIIT Pet

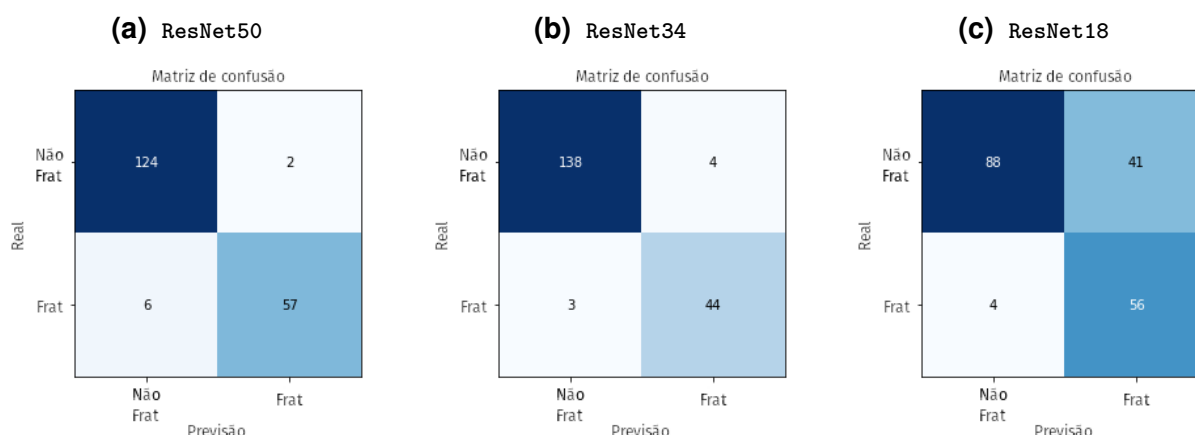
Arquitetura	Tempo de Inferência (ms)			Acurácia
	Jetson	PC (GPU)	PC (CPU)	
Resnet18	139 $\pm$ 7.27	38.6 $\pm$ 1.04	104 $\pm$ 2.39	99.46 %
Resnet34	167 $\pm$ 3.75	44.5 $\pm$ 1.17	160 $\pm$ 3.94	99.73 %
Resnet50	203 $\pm$ 5.79	60 $\pm$ 0.474	180 $\pm$ 3.09	99.66 %

Fonte: Elaborado pelo autor.

mas como é muito similar à versão anexada, somente carregando diretamente o modelo, não será incluído aqui.

3.5.2 Inspeção de rebites

Para este banco de dados, devido ao seu tamanho reduzido e outras características comentadas anteriormente, o processo de treinamento foi relativamente rápido e não ocorreram problemas relacionados à memória disponível.

Figura 15 – Matrizes de confusão para cada arquitetura de RNC avaliada

Fonte: Elaborado pelo autor

Tabela 5 – Tempos de execução - Rebites

Arquitetura	Tempo de Inferência			Acurácia
	Jetson	PC (GPU)	PC (CPU)	
Resnet18	104 $\pm$ 1.43 ms	26.9 $\pm$ 1.44 ms	50.9 $\pm$ 2.15	76.19 %
Resnet34	130 $\pm$ 1.95 ms	30.3 $\pm$ 0.934	78.4 $\pm$ 3.77	96.30 %
Resnet50	147 $\pm$ 2.57 ms	36.4 $\pm$ 0.739 ms	90.6 $\pm$ 7.63	95.77 %

Fonte: Elaborado pelo autor.

Novamente, é visível na tabela 5 que a acurácia da arquitetura *resnet34* se sobressaiu levemente a da *resnet50*, mas neste caso a diferença é tão próxima que, caso se treine os modelos novamente, talvez os resultado se invertam devido às

variáveis aleatórias. Em função do custo computacional do modelo `resnet50`, neste caso sua aplicação não seria ideal, pois os resultados são praticamente idênticos para um modelo mais complexo. É relevante destacar a grande quantidade de falsos positivos para o modelo `resnet18`, apresentado na figura 15c, mas não falsos negativos.

Similar à aplicação anterior, pode-se observar no anexo B, o código utilizado no processo de treinamento da rede neural.

Será discutido subsequentemente em 3.5.3 a relação dos tempos de inferência com a demanda proposta em (STIVANELLO; MASSON; STEMMER, 2022).

3.5.3 Discussão dos Resultados

Como comentado anteriormente, esta seção se propõe a explorar os resultados obtidos em relação às características levantadas anteriormente e aos artigos correlatos, além de também trazer outros pontos gerais sobre o desenvolvimento desta avaliação.

Primeiramente, sobre o uso de memória e dos recursos, no caso do banco de dados *Oxford-IIIT Pet*, durante o processo de inferência no sistema embarcado Jetson foi observado um grande uso de memória, a ponto de extravasar para sua memória virtual (*swap*). Isto não é o ideal, pois reduz drasticamente a performance de uma aplicação devido à latência envolvida no acesso de memória não-volátil, em vez da RAM. No caso da aplicação dos rebites, coube confortavelmente dentro da memória disponível. Felizmente nenhum dos modelos foi completamente inviabilizado devido à memória insuficiente.

Sobre os tempos de inferência, em todos os casos a Jetson tomou mais tempo em comparação ao PC:

- No caso da aplicação dos rebites consistentemente em torno de 4 vezes mais tempo.
- No caso do banco de dados *Oxford-IIIT Pet* em torno de 3.5 vezes mais, mesmo tendo problemas com o uso de memória.

Relacionando com a demanda proposta no artigo (STIVANELLO; MASSON; STEMMER, 2022), de inferência na taxa de 4 à 8 Hz, caso fosse usado a placa Jetson diretamente com arquiteturas propostas, no caso mais difícil de 8 Hz⁹ somente a arquitetura `resnet18`, que possui alta taxa de erro e não seria ideal, atende diretamente a demanda. Na condição de produção mais lenta, de 4Hz¹⁰, todas as arquiteturas propostas atendem ao tempo de execução com folga. Como não houve um requisito de

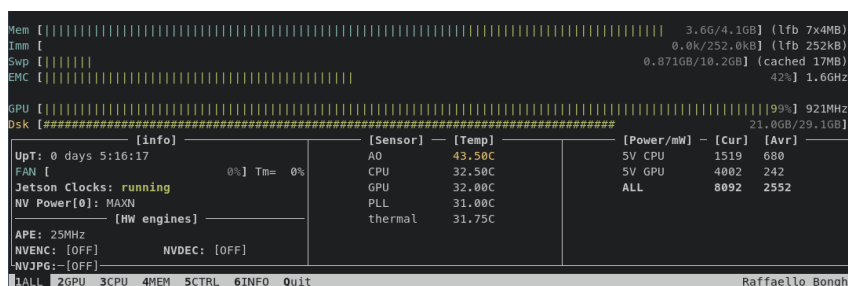
⁹ 125 ms

¹⁰ 250 ms

tempo específico na aplicação *Pets*, então não é tão significativo o tempo de inferência. Porém, comparando seus valores absoluto com a aplicação dos rebites, mesmo no seu menor modelo, *resnet18*, o tempo de execução foi maior que do maior modelo, *resnet50*, da outra aplicação, o que indica a complexidade do sistema.

Prosseguindo para a questão da demanda por potência e ventilação, durante a inferência das aplicações, somente no caso da aplicação *Pets* a temperatura média dos elementos de processamento da placa Jetson consistentemente se mantiveram acima de 40 °C, como mostrado na figura 16, devido ao extenso tempo com a GPU em 100% de utilização necessário para cada inferência.

Figura 16 – Exemplo das estatísticas fornecidas pela ferramenta *jtop* durante o processo de inferência



Fonte: do autor.

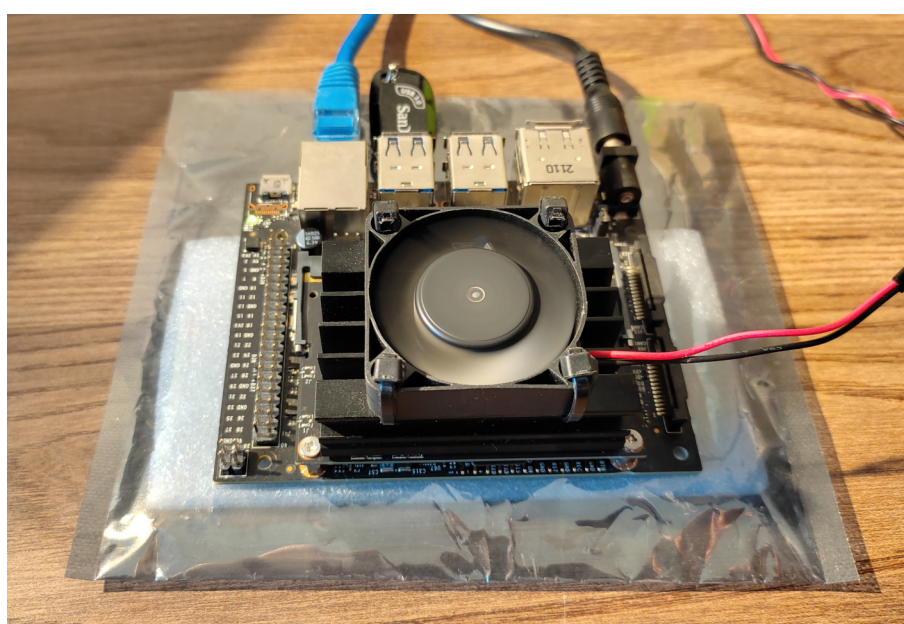
Utilizando uma estimativa simplificada do consumo de energia para cada um dos sistemas avaliados:

- Para a plataforma PC, foi estimado um consumo médio de energia baseada no valor do TDP (*Thermal Dissipation Profile*) dos componentes principais do sistema, de 65 W para CPU e 75W para a GPU. No caso da aplicação utilizando a GPU, foi aplicado um fator de carga de 0,7 sobre este valor, resultando em 120 W. Para o caso somente a CPU, como não houve redução de performance devido a saturação térmica (*thermal throttling*), foi utilizado o valor nominal do TDP, de 65 W. Também é relevante ressaltar que o valor real do consumo de energia de um computador depende de vários outros componentes não considerados, como discos e periféricos e, em geral, tende a ser maior, como por exemplo o estimado nesta ferramenta, que estima em torno de 211 W para potência de carga da configuração utilizada.
- Para a placa NVIDIA Jetson NANO, este valor é explicitado em sua folha de dados, incluída no anexo A, e é em torno de 5-10 W. Foi considerado o valor máximo de 10W para esta estimativa, pois foi utilizada a alimentação P4 que permite a placa utilizar até seu teto de consumo.

Considerando estes dados, foi estimado que a placa NVIDIA Jetson NANO é em torno de 2 a 3 vezes mais eficiente para inferência em comparação à GPU+CPU ou somente CPU. Ou seja, mesmo tomando mais tempo para executar a mesma tarefa, seu uso de energia¹¹ absoluto ainda é menor que da plataforma de comparação, o que pode ser um diferencial em certas aplicações.

Relacionado à questão de potência, para ventilação a placa Jetson demanda somente um pequeno *cooler* para deslocar ar sobre seu dissipador de calor, como demonstrado na figura 17, e que como o volume de ar deslocado é pequeno, mesmo no caso de um ambiente insalubre, seria viável o uso de filtros.

Figura 17 – Placa NVIDIA Jetson NANO com *cooler* acoplado



Já no caso de um PC, comparativamente, a não ser que tenha sido pensado para esta tarefa, demanda o deslocamento de uma grande quantidade de ar para ser devidamente resfriado, o que seria mais difícil de se filtrar adequadamente. Como citado anteriormente em 3.1.4, existem no mercado sistemas de resfriamento completamente passivos (sem ventoinhas), mas em geral tendem a possuir aplicabilidade reduzida¹².

Sobre a última característica levantada, da dimensão física, pode-se dizer que em geral os sistemas embarcados tendem a ocupar menos espaço em comparação a um sistema embarcado. Por exemplo, todo o sistema da Jetson ocupa um volume de aproximadamente 85x100x50 mm ou 425 cm³ incluindo o cooler, enquanto somente o *socket* da CPU de um PC comum já ocupa um volume similar. Qualitativamente, PCs *tendem a* ocupar mais espaço do que sistemas embarcados.

¹¹ [Watt] · [segundo] = [Joule]

¹² devido ao grande volume físico, peso e custo

Agora adentrando no tópico da usabilidade do sistema em si, durante a elaboração do projeto, grande parte do tempo despendido nesta avaliação foi no sentido de resolver dificuldades técnicas, algumas delas da plataforma em si, e várias outras do estado atual da linguagem `python` e de suas bibliotecas. A implementação do sistema de aprendizado de máquina e o uso das ferramentas relacionadas foi relativamente simples e organizado.

Já na parte da plataforma embarcada, o problema encontrado foi que mesmo o kit de desenvolvimento utilizado sendo bem robusto da questão do *hardware* em si, a fabricante somente provém versões defasadas das ferramentas muito utilizadas atualmente para aprendizado de máquina.

Na questão do ecossistema da linguagem de programação `python`, no presente momento da elaboração deste trabalho, é considerado problemático, pois a interoperabilidade das versões anteriores do interpretador com bibliotecas muitas vezes não é documentada, ou sequer é verificada. Uma ferramenta criada para tentar resolver ou minimizar este problema, o versionamento semântico, não é aplicado por várias bibliotecas e a até a linguagem em si, que “Sob este esquema, os números de versão e a forma como eles mudam transmitem o significado do código subjacente e o que foi modificado de uma versão para a próxima” (WERNER, 2018).

Por último, sobre o aspecto financeiro, no momento da elaboração deste trabalho, a placa NVIDIA Jetson NANO com 4GB de RAM está cotada no mercado brasileiro em torno de 3.500,00 BRL. Este preço está inflacionado devido à limitações de estoque e a situação global atual, e, seu preço cotado em lojas no exterior é em torno de USD 100, que mesmo após impostos de importação¹³ e conversão¹⁴ ainda sai por em torno de 827 BRL.

No caso da plataforma PC, o valor mínimo estimado para o computador utilizado é de em torno de 3000 à 3500 reais, mesmo quando estimado utilizando preços em dólar.

Desta forma, caso o preço da plataforma Jetson não estivesse tão alterado, seu custo benefício seria altamente atraente para diversas aplicações compatíveis, além das outras vantagens citadas anteriormente.

¹³ 60 % sobre o valor original.

¹⁴ 1 USD = 5.17 BRL.

4 CONCLUSÕES E TRABALHOS FUTUROS

No presente trabalho foram desenvolvidos e adaptados algoritmos de aprendizado de máquina baseados em RNCs, de forma estudar a viabilidade, performance e usabilidade de sistemas embarcados para esta tarefa.

Durante o desenvolvimento desta avaliação, foi possível executar, com sistemas embarcados, tarefas normalmente delegadas somente à servidores remotos e computadores x86_64, utilizando ferramentas modernas de aprendizado de máquina e abrindo a possibilidade da implementação de sistemas inteligentes independentes da internet com baixo *footprint*.

Durante a elaboração desta avaliação, foram estudados diversos ecossistemas de ferramentas diferentes, precisando ser ponderado cada um de modo a poder utilizá-los e combiná-los efetivamente. Da parte dos sistemas embarcados, se estudou as ferramentas e particularidades do sistema escolhido e como isto integra com outros ecossistemas existentes. Já da parte das ferramentas de aprendizado de máquina, foi necessário um grande esforço no sentido de encontrar versões compatíveis com a arquitetura do sistema embarcado.

Sobre a usabilidade da plataforma escolhida, NVIDIA Jetson NANO, percebe-se que possui grande potencial, por apresentar uma configuração de *hardware* interessante, mas por enquanto sua usabilidade é limitada por seu *software* defasado. O uso e implantação de sistemas envolvendo RNC é possível, mas no momento da elaboração deste trabalho não se integrou bem com as ferramentas e bibliotecas modernas e demandou bastante esforço de configuração para obter resultados efetivos. Apesar das dificuldades técnicas encontradas, todas as combinação de arquiteturas para RNC e banco de dados propostas para avaliação foram bem sucedidas, obtendo tempos de execução razoáveis, mesmo quando sobrecarregando a memória do sistema embarcado, e se provando possível o uso desta tecnologia nesta plataforma.

Como possível primeiro trabalho futuro, sugere-se explorar a viabilização da aplicação dos rebites para inferência local. Estudando métodos para reduzir seu tempo de processamento, possivelmente utilizando alguma forma de otimização, como citado em 2.1.2.4. Além disso, como a placa NVIDIA Jetson NANO possui pinos de entrada e saída, poderia-se pensar em atuar diretamente no controle do descarte das peças defeituosas.

Como segunda possível melhoria, seria ao invés de comparar um sistema embarcado com um PC comum, que tende a não prezar tanto por eficiência energética, comparar com alguma forma de PC industrial, pensado para uso neste tipo de ambiente, e observar se é viável esta implementação e, se sim, qual a diferença de eficiência e usabilidade desta plataforma no ambiente fabril e comercial.

Um último possível aprimoramento sugerido, seria simplesmente coletar mais pontos de dados sobre todo o processo. Especificamente sobre o consumo energético; seria de interesse medir o consumo instantâneo das plataformas avaliadas, não dependendo de estimativas de consumo, mas avaliar quanta energia *realmente* foi gasta, instante a instante.

Considerando o objetivo geral e específico deste trabalho, é possível afirmar que foram bem sucedidos, fornecendo um breve panorama do estado do ecossistema de *software* embarcado no que concerne sua intersecção com inteligência artificial, às tecnologias e complexidades envolvidas e normalmente não consideradas, além de possíveis soluções e oportunidades de aplicação encontradas.

REFERÊNCIAS

- BRIGNOLI, R. Estudo de redes neurais convolucionais e sua aplicação para classificação de ambientes. Florianópolis, p. 55, 2021. 13, 29
- CONDA. *Conda: Package, dependency and environment management for any language Python, R, Ruby, Lua, Scala, Java, JavaScript, C/ C++, Fortran, and more.* 2022. Disponível em: <https://docs.conda.io/en/latest/>. Acesso em: 4 jul 2022. 40
- DEBUGGING TISSUE. *How to save fastai plots when using an external server or terminal environment.* 2020. Disponível em: <https://www.debuggingtissue.com/latest-articles/how-to-save-fastai-plots-when-using-an-external-server-or-terminal-environment>. Acesso em: 4 jul 2022. 37
- FAST.AI. *Installing FastAI.* 2022. Disponível em: <https://docs.fast.ai/#Installing>. Acesso em: 4 jul 2022. 38, 40
- GRAVES, A.; MOHAMED, A.-r.; HINTON, G. Speech recognition with deep recurrent neural networks. 2013. 17
- HONG, C.-H.; VARGHESE, B. Resource management in fog/edge computing: A survey on architectures, infrastructure, and algorithms. *ACM computing surveys*, ACM, Baltimore, v. 52, n. 5, p. 1–37, 2019. ISSN 0360-0300. 13, 26
- HOWARD, J.; GUGGER, S. Fastai: A layered api for deep learning. *Information (Basel)*, Cornell University Library, arXiv.org, Ithaca, v. 11, n. 2, p. 108, 2020. ISSN 2078-2489. 33
- KIM, J.; CHO, J. Low-cost embedded system using convolutional neural networks-based spatiotemporal feature map for real-time human action recognition. *Applied sciences*, MDPI AG, Basel, v. 11, n. 11, p. 4940, 2021. ISSN 2076-3417. 29
- KIM, Y. Convolutional neural networks for sentence classification. 2014. 17
- KOVACS, Z. L. *Redes neurais artificiais: fundamentos e aplicações: um texto básico.* 2. ed. São Paulo: collegium cognitio, 1996. 174 p. ISBN 85-86396-02-08. 15, 16
- LEE, J. *et al.* Quantune: Post-training quantization of convolutional neural networks using extreme gradient boosting for fast deployment. *Future generation computer systems*, Elsevier B.V, v. 132, p. 124–135, 2022. ISSN 0167-739X. 22
- LEMOES, Y. M. V. Inspeção automática de defeitos em ovos comerciais usando visão computacional. Florianópolis, p. 1–170, 2021. 19, 29
- MCCULLOCK, W.; PITTS, W. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 1943. 15
- PARKHI, O. M.; VEDALDI, A.; JAWAHAR, A. Z. C. V. Cats and dogs. *Conference on Computer Vision and Pattern Recognition*, IEEE, p. 1–8, 2012. Disponível em: <https://www.robots.ox.ac.uk/~vgg/publications/2012/parkhi12a/parkhi12a.pdf>. Acesso em: 3 Jul 2022. 34, 35, 36

- RAMOS, U.; STIVANELLO, M. E.; STEMMER, M. R. Adaptable architecture for the development of computer vision systems in fpga. *Revista IEEE América Latina*, IEEE, v. 18, n. 12, p. 2104–2111, 2020. ISSN 1548-0992. 30
- SHU, G. *et al.* If-cnn: Image-aware inference framework for cnn with the collaboration of mobile devices and cloud. *IEEE Access*, v. 6, p. 68621–68633, 2018. 13
- STIVANELLO, M. E.; MASSON, J. E. N.; STEMMER, M. R. A cnn approach for online metal can end rivet inspection. *International Journal Computer Applications in Technology*, inderscience, v. 69, 2022. 27, 28, 30, 35, 37, 42
- WANGENHEIM, A. von. **Deep Learning::Reconhecimento de Imagens**. 2017. Disponível em: <https://lapix.ufsc.br/ensino/visao/visao-computacionaldeep-learning/deep-learningreconhecimento-de-imagens/#ResNet>. Acesso em: 23 jun 2022. 18
- WEI, G. *et al.* Demo: Fpga-cloud architecture for cnn. In: *2018 24th Asia-Pacific Conference on Communications (APCC)*. [S.l.: s.n.], 2018. p. 7–8. 13
- WERNER, T. P. *Versionamento Semântico 2.0.0*. 2018. Disponível em: <https://semver.org/lang/pt-BR/>. Acesso em: 1 jul 2022. 38, 45
- YANG, S. *et al.* Image data augmentation for deep learning: A survey. 2022. 36

APÊNDICES

APÊNDICE A – CÓDIGO EXEMPLO PARA Max Pooling

```
11 import sys
12
13 import numpy as np
14 from PIL import Image
15
16 img = Image.open(sys.argv[1])
17
18
19 STRIDE = 3
20
21 src = np.array(img)
22
23 res = np.zeros([int(img.width/STRIDE), int(img.height/STRIDE)], np.uint8)
24 tmp = np.zeros([STRIDE, STRIDE], np.uint8)
25
26 for y in range(0, img.height-STRIDE, STRIDE):
27     for x in range(0, img.width-STRIDE, STRIDE):
28         for i in range(STRIDE):
29             tmp[i:i+1] = src[x:x+1, y:y+STRIDE]
30             res[int(x/STRIDE), int(y/STRIDE)] = int(np.max(tmp))
31
32 res_img = Image.fromarray(res)
33
34 res_img.save(f"{sys.argv[1].split('.')[0]}.max_pool.png", mode="L")
```

APÊNDICE B – *Script* DE TREINAMENTO REBITES

1 Imports

```
[ ]: # Tutorial de como instalar a fastai
# https://stackoverflow.com/questions/57910449/moduleNotFoundError-no-module-named-fastai
from fastai.vision.all import *
import numpy as np
```

2 Configuração

```
[ ]: model = resnet50 # Seleciona arquitetura
```

3 Verificando GPU

```
[ ]: if not torch.cuda:
    print("torch is not using cuda!")
    exit(0)
else:
    print(f"CUDA device: {torch.cuda.get_device_name()}")
```

4 Define Caminhos

```
[ ]: # Atualizar o caminho do dataset
path = Path(".");
files = get_image_files(path/"images")

f"input file count: {len(files)}"
```

5 Cria Dataloader

```
[ ]: def valid(f):
    """
    label function
    True = fractured
    """
    return (f[0] == 'f')

np.random.seed(42)
data = ImageDataLoaders.from_name_func(path, files, valid, valid_pct=0.2) # 20%
    ↳ para validação

#ds_tfms=get_transforms(do_flip=True, flip_vert=True), size=224, bs=16 ,
    ↳ num_workers=4).normalize(imagenet_stats)
```

```
#Explicação do data augmentation: https://forums.fast.ai/t/how-does-get-transforms-work/55497/10
```

```
#data.classes
```

```
[ ]: data.show_batch()
```

```
[ ]: #data.classes, data.c, len(data.train_ds), len(data.valid_ds)
```

5.1 Treinando o modelo

```
[ ]: learn = cnn_learner(data, model, metrics=error_rate)
```

```
[ ]: # Procura a melhor taxa de aprendizagem, não achei que ajudou muito  
learn.lr_find()
```

```
#learn.recorder.plot() # deprecado
```

```
[ ]: #Estágio 1, apenas a camada final de decisão é treinada  
learn.fit_one_cycle(5)#, max_lr=slice(1e-4,1e-2))
```

```
[ ]: learn.save(f'{model.__name__}-stage-1')
```

```
[ ]: #Descongela a rede para um treinamento de todas as camadas  
learn.unfreeze()
```

```
[ ]: # Procura a melhor taxa de aprendizagem, não achei que ajudou muito  
# Caso o plot não apareça, você pode tentar adicionar um início e fim  
#learn.lr_find(start_lr=1e-5, end_lr=1e-1)  
  
# learn.lr_find() # Não funciona: "IndexError: index 0 is out of bounds for "  
↪dimension 0 with size 0"  
  
# learn.recorder.plot() #deprecado
```

```
[ ]: learn.fit_one_cycle(5) #, max_lr=slice(1e-5,1e-1))
```

```
[ ]: learn.save(f'{model.__name__}-stage-2')
```

5.2 Interpretação

```
[ ]: learn.load(f'{model.__name__}-stage-1');
```

```
[ ]: interp = ClassificationInterpretation.from_learner(learn)
```

```
[ ]: interp.plot_confusion_matrix()
```

6 Secciona Imagens para teste

```
[ ]: valid_img = list(filter(lambda x: not valid(x.name), files))

f"valid images count: {len(valid_img)}"
```

```
[ ]: %%timeit -r 10 -n 100
[learn.predict(valid_img[0])]
```

```
[ ]: frrat_count = 0;
nfrat_count = 0;

i=0
for img_path in files:
    print(f"{{(100.0*i)/len(files)}} %", end="")

    result = str(learn.predict(img_path)[0]);
    if result == 'True':
        frrat_count += 1
    else:
        nfrat_count += 1
    i+=1
```

```
[ ]: print(f"factured count: {frat_count} not fractured count: {nfrat_count}")
print(f"expected fractured: {len(files)-len(valid_img)}, expected not fractured:
↪ {len(valid_img)}")
```

APÊNDICE C – *Script* DE TREINAMENTO *OxfordII Pet*

1 Preparação

1.1 Imports

```
[ ]: # Tutorial de como instalar a fastai
# https://stackoverflow.com/questions/57910449/moduleNotFoundError-no-module-named-fastai
from fastai.vision.all import *
import numpy as np
```

1.2 Configuração

Seleciona arquitetura e tamanho de bloco, para Resnet50, o bs padrão falha (4Gb Ram)

```
[ ]: MODEL, BS = (resnet50, 24)
#MODEL, BS = (resnet34, 64)
#MODEL, BS = (resnet18, 64)
```

1.3 Verificando GPU

```
[ ]: if not torch.cuda:
    print("torch is not using cuda!")
    exit(0)
else:
    print(f"CUDA device: {torch.cuda.get_device_name()}")
```

2 Preparação Modelo

2.1 Carrega Imagens

```
[ ]: # Atualizar o caminho do dataset
path = untar_data(URLs.PETS)
files = get_image_files(path/"images")

print(f"input file count: {len(files)}")
```

2.2 Cria Dataloader

```
[ ]: def label_func(f):
    return f[0].isupper()

np.random.seed(42)

dls = ImageDataLoaders.from_name_func(path, files, label_func,
    item_tfms=Resize(224), bs=BS) # default bs=64

[ ]: dls.show_batch()
```

```
#Explicação do data augmentation: https://forums.fast.ai/t/how-does-get-transforms-work/55497/10
```

```
#data.classes
```

```
[ ]: data.show_batch()
```

```
[ ]: #data.classes, data.c, len(data.train_ds), len(data.valid_ds)
```

5.1 Treinando o modelo

```
[ ]: learn = cnn_learner(data, model, metrics=error_rate)
```

```
[ ]: # Procura a melhor taxa de aprendizagem, não achei que ajudou muito  
learn.lr_find()
```

```
#learn.recorder.plot() # deprecado
```

```
[ ]: #Estágio 1, apenas a camada final de decisão é treinada  
learn.fit_one_cycle(5)#, max_lr=slice(1e-4,1e-2))
```

```
[ ]: learn.save(f'{model.__name__}-stage-1')
```

```
[ ]: #Descongela a rede para um treinamento de todas as camadas  
learn.unfreeze()
```

```
[ ]: # Procura a melhor taxa de aprendizagem, não achei que ajudou muito  
# Caso o plot não apareça, você pode tentar adicionar um início e fim  
#learn.lr_find(start_lr=1e-5, end_lr=1e-1)  
  
# learn.lr_find() # Não funciona: "IndexError: index 0 is out of bounds for "  
↪dimension 0 with size 0"  
  
# learn.recorder.plot() #deprecado
```

```
[ ]: learn.fit_one_cycle(5) #, max_lr=slice(1e-5,1e-1))
```

```
[ ]: learn.save(f'{model.__name__}-stage-2')
```

5.2 Interpretação

```
[ ]: learn.load(f'{model.__name__}-stage-1');
```

```
[ ]: interp = ClassificationInterpretation.from_learner(learn)
```

```
[ ]: interp.plot_confusion_matrix()
```

6 Secciona Imagens para teste

```
[ ]: valid_img = list(filter(lambda x: not valid(x.name), files))

f"valid images count: {len(valid_img)}"
```

```
[ ]: %%timeit -r 10 -n 100
[learn.predict(valid_img[0])]
```

```
[ ]: frrat_count = 0;
nfrat_count = 0;

i=0
for img_path in files:
    print(f"{{(100.0*i)/len(files)}} %", end="")

    result = str(learn.predict(img_path)[0]);
    if result == 'True':
        frrat_count += 1
    else:
        nfrat_count += 1
    i+=1
```

```
[ ]: print(f"factured count: {frat_count} not fractured count: {nfrat_count}")
print(f"expected fractured: {len(files)-len(valid_img)}, expected not fractured:
↪ {len(valid_img)}")
```

ANEXOS

ANEXO A – PÁGINA DE DADOS GERAIS NVIDIA JETSON NANO



DATA SHEET

NVIDIA Jetson Nano System-on-Module

Maxwell GPU + ARM Cortex-A57 + 4GB LPDDR4 + 16GB eMMC

Maxwell GPU[◊]

128-core GPU | End-to-end lossless compression | Tile Caching | OpenGL[®] 4.6 | OpenGL ES 3.2 | Vulkan™ 1.1 | CUDA[®] | OpenGL ES Shader Performance (up to): 512 GFLOPS (FP16) | Maximum Operating Frequency: 921MHz

CPU

ARM[®] Cortex[®]-A57 MPCore (Quad-Core) Processor with NEON Technology | L1 Cache: 48KB L1 instruction cache (I-cache) per core; 32KB L1 data cache (D-cache) per core | L2 Unified Cache: 2MB | Maximum Operating Frequency: 1.43GHz

Audio

Industry standard High Definition Audio (HDA) controller provides a multichannel audio path to the HDMI interface.

Memory

Dual Channel | System MMU | Memory Type: 4ch x 16-bit LPDDR4 | Maximum Memory Bus Frequency: 1600MHz | Peak Bandwidth: 25.6 GB/s | Memory Capacity: 4GB

Storage

eMMC 5.1 Flash Storage | Bus Width: 8-bit | Maximum Bus Frequency: 200MHz (HS400) | Storage Capacity: 16GB

Boot Sources

eMMC and USB (recovery mode)

Networking

10/100/1000 BASE-T Ethernet | Media Access Controller (MAC)

Imaging

Dedicated RAW to YUV processing engines process up to 1400Mpix/s (up to 24MP sensor) | MIPI CSI 2.0 up to 1.5Gbps (per lane) | Support for x4 and x2 configurations (up to four active streams).

Operating Requirements

Temperature Range (T_j): -25 – 97C* | Module Power: 5 – 10W | Power Input: 5.0V

Display Controller

Two independent display controllers support DSI, HDMI, DP, eDP:
 MIPI-DSI (1.5Gbps/lane): Single x2 lane | Maximum Resolution: 1920x960 at 60Hz (up to 24bpp)
 HDMI 2.0a/b (up to 6Gbps) | DP 1.2a (HBR2 5.4 Gbps) | eDP 1.4 (HBR2 5.4Gbps) | Maximum Resolution (DP/eDP/HDMI): 3840 x 2160 at 60Hz (up to 24bpp)

Clocks

System clock: 38.4MHz | Sleep clock: 32.768kHz | Dynamic clock scaling and clock source selection

Multi-Stream HD Video and JPEG

Video Decode

H.265 (Main, Main 10): 2160p 60fps | 1080p 240fps
 H.264 (BP/MP/HP/Stereo SEI half-res): 2160p 60fps | 1080p 240fps
 H.264 (MVC Stereo per view): 2160p 30fps | 1080p 120fps
 VP9 (Profile 0, 8-bit): 2160p 60fps | 1080p 240fps
 VP8: 2160p 60fps | 1080p 240fps
 VC-1 (Simple, Main, Advanced): 1080p 120fps | 1080i 240fps
 MPEG-2 (Main): 2160p 60fps | 1080p 240fps | 1080i 240fps

Video Encode

H.265: 2160p 30fps | 1080p 120fps
 H.264 (BP/MP/HP): 2160p 30fps | 1080p 120fps
 H.264 (MVC Stereo per view): 1440p 30fps | 1080p 60fps
 VP8: 2160p 30fps | 1080p 120fps

JPEG (Decode and Encode): 600 MP/s

Peripheral Interfaces

xHCI host controller with integrated PHY: 1 x USB 3.0, 3 x USB 2.0 | USB 3.0 device controller with integrated PHY | EHCI controller with embedded hub for USB 2.0 | 4-lane PCIe: one x1/2/4 controller | single SD/MMC controller (supporting SDIO 4.0, SD HOST 4.0) | 3 x UART | 2 x SPI | 4 x I2C | 2 x I2S: support I2S, RJM, LJM, PCM, TDM (multi-slot mode) | GPIOs

Mechanical

Module Size: 69.6 mm x 45 mm | PCB: 8L HDI | Connector: 260 pin SO-DIMM

Note: Refer to the software release feature list for current software support; all features may not be available for a particular OS.

[◊] Product is based on a published Khronos Specification and is expected to pass the Khronos Conformance Process. Current conformance status can be found at www.khronos.org/conformance.

* See the *Jetson Nano Thermal Design Guide* for details. Listed temperature range is based on module T_j characterization.