UNIVERSIDADE FEDERAL DE OURO PRETO
INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS
DEPARTAMENTO DE COMPUTAÇÃO

FREDERICO LUIZ MARTINS DE SOUSA
Orientador: Prof. Dr. Ricardo Augusto Rabelo Oliveira
Coorientador: Prof. Msc. Mateus Coelho Silva

USO DE SLAM NA CRIAÇÃO DE MODELOS PARA ODOMETRIA VISUAL BASEADA EM DEEP LEARNING PARA ROBÓTICA MÓVEL

Ouro Preto, MG
2021

UNIVERSIDADE FEDERAL DE OURO PRETO
INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS
DEPARTAMENTO DE COMPUTAÇÃO

FREDERICO LUIZ MARTINS DE SOUSA

USO DE SLAM NA CRIAÇÃO DE MODELOS PARA ODOMETRIA VISUAL BASEADA EM DEEP LEARNING PARA ROBÓTICA MÓVEL

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal de Ouro Preto como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Ricardo Augusto Rabelo Oliveira
Coorientador: Prof. Msc. Mateus Coelho Silva

Ouro Preto, MG
2021

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE OURO PRETO
REITORIA
INSTITUTO DE CIENCIAS EXATAS E BIOLOGICAS
DEPARTAMENTO DE COMPUTACAO

FOLHA DE APROVAÇÃO

Frederico Luiz Martins de Sousa

Uso de Slam na Criacao de Modelos Para Odometria Visual Baseada e Deep Learning para robotica movel

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal
de Ouro Preto como requisito parcial para obtenção do título de Bacharel em Ciência da Computação

Aprovada em 5 de Janeiro de 2022.

Membros da banca

Ricardo Augusto Rabelo Oliveira (Orientador) - Doutor - Universidade Federal de Ouro Preto
Saul Emanuel Delabrida Silva (Examinador) - Doutor - Universidade Federal de Ouro Preto
Carlos Frederico M. da Cunha Cavalcanti (Examinador) - Doutor - Universidade Federal de Ouro Preto

Ricardo Augusto Rabelo Oliveira, Orientador do trabalho, aprovou a versão final e autorizou seu depósito na Biblioteca Digital de Trabalhos de Conclusão de Curso da UFOP em 05/01/2022.

Documento assinado eletronicamente por **Ricardo Augusto Rabelo Oliveira**, **PROFESSOR DE MAGISTERIO SUPERIOR**, em 11/01/2022, às 21:46, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site http://sei.ufop.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0 , informando o código verificador **0265674** e o código CRC **4DB2DE8C**.

Referência: Caso responda este documento, indicar expressamente o Processo nº 23109.013468/2021-94 SEI nº 0265674

R. Diogo de Vasconcelos, 122, - Bairro Pilar Ouro Preto/MG, CEP 35400-000
Telefone: 3135591692 - www.ufop.br

Dedico este trabalho aos meus pais, meu irmão e minha família. Jamais chegaria tão longe sem vocês.

Agradecimentos

Primeiramente, eu agradeço aos meus pais João Luís e Denise, pelo apoio incondicional e por me levantar nos momentos mais difíceis da minha vida. Agradeço ao meu irmão João Victor por me aguentar reclamando da vida acadêmica e pessoal e sempre com paciência para que eu não deixasse os problemas me dominarem. Também agradeço a toda minha família por todo o apoio, em especial meus avós Geraldo e Vilma e meus primos Enrico e Tomás. Agradeço também aos meus primos Mauro e Cleize por todos os incentivos, não somente acadêmicos, desde minha infância, muito obrigado por tudo! Agradeço à minha querida escola na qual eu estudei por toda a minha vida por oferecer as bases educacionais para que eu pudesse estar aqui, à Escola Estadual Lopes Franco, meu muito obrigado! Na universidade eu tive a oportunidade de ter várias boas pessoas que me ajudaram chegar onde eu cheguei e elas passam por três lugares fundamentais na minha caminhada. Esses lugares são: as pessoas da minha casa, o laboratório iMobilis e o laboratório ITV. O meu agradecimento ao grande Hugo Ziviani que desde o primeiro período me motivou a nunca desistir e me ajudou sempre da melhor forma nos caminhos acadêmicos, muito obrigado e estaremos sempre juntos! Agradeço aos meus grandes amigos de casa, Gabriel Matoso e Felipe Silva, os quais eu sempre levarei no coração com muito carinho por todas as matérias que fizemos juntos e por tudo que me ensinaram sobre convivência, muito obrigado por existirem! Agradeço a todos os membros do laboratório iMobilis, em especial Mateus Coelho, Ricardo Câmara, Maurício Silva, Thiago D'Angelo e Débora Lage, por todas as tardes de muito aprendizado e muita zoeira acompanhados de um pagode dos anos 90. Um obrigado especial ao Mateus Coelho que sempre me orientou e teve toda a paciência do mundo pra me explicar todos os conceitos (do mais simples ao mais complexo) em que eu tinha dúvidas, eu tenho certeza que você será um grande professor! Agradeço também ao professor Ricardo Rabelo pela orientação e pelas muitas oportunidades de aprendizado durante o curso. Agradeço também aos grandes amigos que fiz no laboratório ITV, onde tive contato com pessoas extraordinárias e experiências que guardarei pra sempre no coração. Em especial aos grandes companheiros da vida pessoal e acadêmica Jacó, Bidas, Nazário, Maurício e Mário. Muito obrigado por toda a força que vocês já me deram! Além das grandes orientações de Héctor, Felipe, Buneku, Lekin, Speto, Levi e outros gigantes que eu sempre serei grato! Também gostaria de agradecer grandes amigos que tive contato no DECOM que são eles: Marcelo Edivan, Ricardo David, Tiago Sardi, Anderson e tantos outros que eu posso agradecer por sempre me ajudarem e fazerem uma parte grande dessa etapa da minha vida. Todos que eu mencionei (e posso ter esquecido alguns) terão sempre um lugar especial no meu coração e minha gratidão eterna justamente por ser quem vocês são! Muito obrigado por tudo, todos vocês!

[...] As nossas posturas, a nossa suposta auto importância, a ilusão de termos qualquer posição de privilégio no Universo, são desafiadas por este pontinho de luz pálida. O nosso planeta é um grão solitário na imensa escuridão cósmica que nos cerca. Na nossa obscuridade, em toda esta vastidão, não há indícios de que vá chegar ajuda de outro lugar para nos salvar de nós próprios. [...]

[...] Para mim, destaca a nossa responsabilidade de sermos mais amáveis uns com os outros, e para preservarmos e protegermos o "pálido ponto azul", o único lar que conhecemos até hoje.

- Carl Sagan - Pálido Ponto Azul (SAGAN, 1994)

Resumo

A odometria é um problema comum em sistemas de navegação onde se tem a necessidade de estimar a posição do veículo ou dispositivo móvel no ambiente. Para realizar tarefas autônomas, dispositivos robóticos ou inteligentes precisam de ter ciência de sua posição no ambiente. Existem várias estratégias para resolver o problema de odometria, neste trabalho nós exploramos uma solução de odometria visual através de redes neurais profundas para inferência da posição do veículo robótico no ambiente conhecido e mapeado. Para realizar esta tarefa, um primeiro robô equipado com um LIDAR, IMU e câmera mapeia o ambiente através de uma técnica de SLAM. Os dados capturados pelo primeiro robô são usados como ground truth para treinar uma rede neural e posteriormente, outros robôs com apenas uma câmera monocular podem se localizar no ambiente. Também propomos uma validação e avaliação das redes neurais treinadas. A avaliação se baseia em três etapas: uma avaliação e validação de possíveis arquiteturas de redes neurais, comparação da arquitetura treinada *ResNet* com o algoritmo da literatura Orb-Slam e por fim uma avaliação de desempenho da rede treinada em três sistemas embarcados (Raspberry Pi 4, Jetson Nano e Jetson TX2).

Palavras-chave: Robótica móvel. Odometria. Computação em edge. Redes neurais profundas. Robot Operating System.

Abstract

Odometry is a common problem in navigation systems where there is a need to estimate the position of the vehicle or carrier in the environment. To perform autonomous tasks, robotic or intelligent devices need to be aware of its position in the environment. There are many strategies to solve an odometry problem, in this work we explore a visual odometry solution with a deep neural network to infer the robotic vehicle's position in a known and mapped environment. To perform this task the first robot, equipped with a LIDAR, IMU, and camera, map the environment through a SLAM technique. The data gathered by this first robot is used as ground truth to train the neural network and later, other robots with only one camera can locate themselves in the environment. We also propose a validation and evaluation of the neural network. The evaluation is based on three steps: an evaluation and validation of possible neural network architectures, comparison of the trained *ResNet* architecture with the Orb-Slam literature algorithm and, finally, an performance evaluation of the trained network in three embedded systems (Raspberry Pi 4, Jetson Nano and Jetson TX2).

Keywords: mobile robotics, odometry, edge-computing, deep neural networks, Robot Operating System.

Lista de Ilustrações

Lista de Tabelas

Lista de Algoritmos

HectorSlam - Algoritmo de SLAM (mapeamento e localização simultâneos) disponível de forma aberta no ROS (Robot Operating System).

GMapping - Algoritmo de SLAM (mapeamento e localização simultâneos) disponível de forma aberta no ROS (Robot Operating System).

Orb-Slam2 - Algoritmo de SLAM (mapeamento e localização simultâneos) disponível de forma aberta no ROS (Robot Operating System).

Lista de Abreviaturas e Siglas

GPS - Global Positioning System - Sistema de Posicionamento Global

IMU - Inertial Measurement Unit - Unidade de Medição Inercial

RGBD - Red, Green, Blue, Depth - Vermelho, Verde, Azul e Profundidade - Câmeras monoculares coloridas com dados de profundidade

LIDAR - Light Detection and Ranging

SLAM - Simultaneous Localization and Mapping - Mapeamento e localização simultâneos

IA - Inteligência Artificial

DNNs - Deep Neural Networks - Redes Neurais Profundas

IoT - Internet of Things - Internet das Coisas

DL - Deep Learning - Aprendizado Profundo

ROS - Robot Operating System

URDF - Universal Robot Description File - Arquivo de Descrição Universal de Robôs

CNNs - Convolutional Neural Networks - Redes Neurais Convolucionais

FPS - Frames Per Second - Quadros Por Segundo

Sumário

1 Introdução

A odometria é um problema comum em sistemas de navegação onde se tem a necessidade de estimar a posição do veículo ou robô no ambiente. Essa técnica é útil em casos onde se faz necessário a realização de tarefas de navegação autônoma e desvio de obstáculos. Logo, a plataforma robótica precisa manter salvas, as informações de sua posição e orientação no espaço (MOHAMED et al., 2019). No entanto, os problemas de odometria podem ser resolvidos por diferentes estratégias e sensores de medição, sendo a forma mais comum e intuitiva, o uso de GPS's (Global Positioning System) (BENDER; KOCH; CREMERS, 2017), para a localização global do veículo ou dispositivo robótico. Os dispositivos de GPS se baseiam em sinais de satélites (através de ondas de rádio) a fim de extrair a posição final do dispositivo no planeta Terra. Logo, devido a sua natureza, os dispositivos de GPS podem não ser a melhor alternativa para a auto-localização em florestas, cavernas, submerso na água ou até mesmo dentro de prédios e casas. Isso se dá pelo fato de que os sinais de ondas de rádio são facilmente atenuadas por concreto, árvores ou pela água, resultando assim em um baixo sinal de transmissão.

Para os casos de usos citados anteriormente, existem alternativas que podem ser empregadas para determinar a posição e orientação do veículo. As estratégias de medição de odometria mais comuns são listadas a seguir:

- Odometria visual;
- Odometria através de laser;
- Odometria por medição de deslocamento das rodas;
- Odometria por sensores inerciais - IMU (Inertial Measurement Unit);

A odometria visual, de forma geral, se baseia em estimar a posição do dispositivo através da diferença entre imagens e pode ser realizada através de câmeras stereo (que são câmeras de profundidade) (GIUBILATO et al., 2019) ou monoculares (GEIGER; ZIEGLER; STILLER, 2011). As câmeras stereo se baseiam em duas câmeras dispostas em uma distância conhecida que tem como objetivo extrair a profundidade das imagens. Com esse dado de profundidade se pode reconstruir as imagens da câmera em cenários 3D, o que facilita a segmentação de features do ambiente e consequentemente facilita a precisão de um cálculo odométrico. A Figura 1.1 demonstra a reconstrução de duas imagens capturadas por uma câmera stereo em um cenário 3D. As câmeras monoculares são as câmeras com apenas uma lente, ou seja, são câmeras comuns que capturam apenas uma imagem. Também podemos ter estratégias implementadas em câmeras RGBD (Red, Green, Blue, Depth) que são câmeras monoculares coloridas (RGB) mas que contém uma informação de profundidade através de algum sensor externo como um laser.

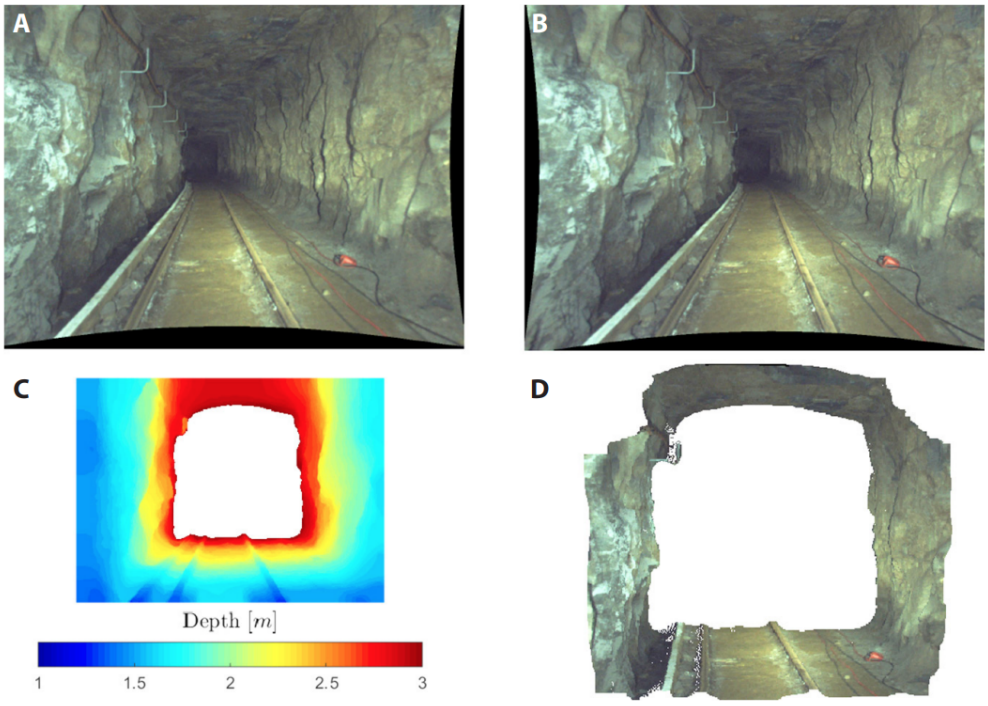
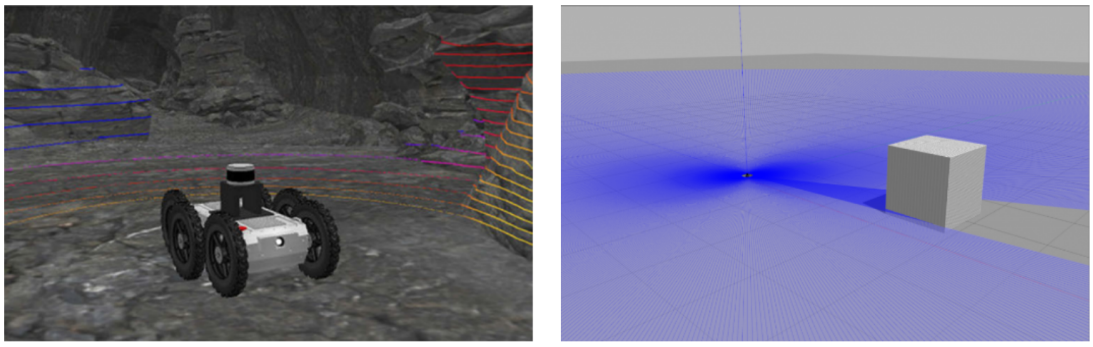


Figura 1.1 – Transformação de duas imagens (A e B) em um ambiente 3D (C e D). Quanto mais vermelho, maior a profundidade da cena. Imagem retirada de Grehl et al. (GREHL et al., 2015)

O tipo de odometria por laser é realizado através da estimativa da posição do robô por características do ambiente que são identificadas por LIDAR's (Light Detection and Ranging) planares ou espaciais (SHAN; ENGLOT, 2018). Os lasers do tipo LIDAR são lasers rotativos que visam escanear os obstáculos de um ambiente. Isso é feito de forma que a medida que um laser gira, também é recebido um feedback da distância na qual esse laser encostou. Fazendo esse processo de forma progressiva e circular, é possível saber quais obstáculos estão posicionados ao redor do laser. Os lasers planares se baseiam em apenas um feixe de luz que escaneia apenas em uma dimensão, por isso o nome planar. Esses lasers escaneiam mapas planares e visualizam somente os objetos que estão em sua altura de leitura. Os lasers espaciais capturam dados nas três dimensões, logo possuem mais de um feixe de luz em seus escaneamentos. A Figura 1.2 demonstra as leituras de lasers planares e espaciais.

A odometria de roda é medida através de sensores embarcados nos motores ou nas rodas do robô que tem o objetivo de medir a velocidade de rotação dos mesmos. Esses sensores calculam o número de revoluções da roda ou do eixo do motor e a partir desse dado, é estimado o quanto o robô se deslocou. Essa estratégia pode ser identificada nos antigos rovers da NASA que embarcaram em missões em Marte (CHENG; MAIMONE; MATTHIES, 2005). Esse tipo de odometria pode sofrer com problemas de deslizamento, onde a roda gira, no entanto o robô não se movimenta. Isso pode ser observado também em situações onde o robô rotaciona em seu próprio eixo. Logo, a precisão dessa estratégia depende diretamente de filtragens dessas movimentações



(a) **(b)**

Figura 1.2 – Exemplo de leitura de lidar espacial em **(a)** (Imagem **(a)** retirada de Azpúrua et al. (AZPURUA et al., 2021)) e exemplo de leitura planar em **(b).** [1]

e eliminação do erro de deslizamento.

A odometria por sensores inerciais (IMU) é calculada através das leituras de aceleração e giroscópio em cada um dos 3 possíveis eixos do robô (LEUTENEGGER et al., 2015). Esse tipo de estratégia sofre com interferências nas medições dos sensores inerciais. Os sensores inerciais também acumulam muito erro com o tempo. A solução nesse caso é de se utilizar filtros de Kallman como o filtro EKF apresentado por Alatise e Hancke (ALATISE; HANCKE, 2017), ou então investir em sensores mais precisos, no entanto mais caros.

Todas as estratégias de odometria já mencionadas contém limitações e um custos associados. Esses custos estão ligados tanto com relação ao financeiro quanto ao custo computacional de processamento. Como mencionado na anteriormente, abordagens de cálculo de odometria utilizando GPS não cobrem todos os casos de uso da odometria. Devido a sua propagação através de sinais de rádio, o sinal do GPS é atenuado dentro de construções, como casas, prédios industriais e cavernas, o que inviabiliza o seu uso nesses ambientes e ambientes relacionados. Outra estratégia adotada na literatura (YAN et al., 2017), é de combinação das diferentes estratégias descritas anteriormente para se ter um cálculo mais refinado da odometria. No entanto, combinar diferentes estratégias pode ter um alto custo tanto financeiro quanto computacional. Logo, se faz necessário um método que resolva a odometria de forma confiável com poucos sensores.

A resolução da odometria visual através de câmeras monoculares se torna relevante pois como apresentado anteriormente, pode ser necessário variados sensores para se ter uma odometria confiável. Ao se utilizar câmeras monoculares, o custo de uma aplicação de cálculo odométrico pode ser razoavelmente reduzido. Além disso, em uma abordagem de odometria visual utilizando aprendizagem profunda é interessante que cada robô tenha capacidade de calcular sua odometria. Isso se torna evidente em ambientes onde se tem múltiplos robôs, onde uma abordagem em nuvem se torna mais onerosa devido ao grande número de dados que teriam de ser trocados. Para

[1] Imagem retirada de https://kiranpalla.com/autonomous-navigation-ros-differential-drive-robot-simulation/creating-map-using-laser-scanner-and-gmapping/. Acesso em 14/12/21.

isso, temos o conceito de inteligência artificial na borda (*Edge-AI*), que será abordado mais a fundo no Capítulo 2.

Nossa abordagem visa um cenário onde múltiplos robôs exploram um ambiente doméstico ou industrial. Para que os robôs possuam dados de odometria e não seja necessário que todos eles possuam diferentes tipos de sensores, propomos uma abordagem onde seria necessário apenas uma câmera monocular. Para realizar essa tarefa, exploramos redes neurais profundas para resolver um problema de odometria visual. Redes neurais profundas são redes neurais artificiais que reproduzem o funcionamento de neurônios biológicos e têm sido utilizadas para resolver vários problemas de visão computacional, como mostrado por Voulodimos et al. (VOULODIMOS et al., 2018). Isso acontece devido a sua alta capacidade de aprender e generalizar problemas, no entanto, redes neurais profundas demandam um elevado número de dados para alimentá-las e melhorar o processo de aprendizado.

Para colher todos os dados necessários, nós exploramos uma abordagem de SLAM (Simultaneous Localization and Mapping) para construir o *ground-truth* da rede neural. Na técnica de SLAM, o robô mapeia e se localiza de forma simultânea dentro do ambiente. Essa técnica pode ser aplicada a robôs de variadas formas, como é mostrado em Taketomi et al. (TAKETOMI; UCHIYAMA; IKEDA, 2017). O *ground-truth* se trata da referência da verdade do que uma rede neural irá aprender.

Neste trabalho, temos o objetivo geral de resolver um problema de odometria visual através de aprendizado profundo e redes neurais. Mais especificamente, aplicamos nossa abordagem em um cenário onde múltiplos robôs explorarão um ambiente previamente mapeado e extraem seus dados de odometria a partir de imagens. A Figura 1.3 mostra o robô desenvolvido para essa aplicação e um segundo robô utilizado para os testes de exploração.

Para realizar esta tarefa, treinamos uma rede neural de regressão que tem como objetivo localizar o veículo ou robô dentro de um ambiente já conhecido e mapeado. No processo de treinamento de nossa rede neural, utilizamos uma função de perda para regressões com o objetivo de reconhecer pontos do mapa gerado pelo SLAM, através de imagens. Usamos o mapa gerado pelo SLAM como o *ground truth*, relacionando uma imagem com uma dada posição e orientação dentro do mapa. Então, a rede neural de regressão aprende que um frame está relacionado com uma posição em uma mapa 2D.

Essa abordagem pode ser aplicada a um caso de uso onde existem três ou mais robôs. Portanto, pelo menos um desses robôs deve estar equipado com um LIDAR, IMU e uma câmera monocular. Esse primeiro robô explora o ambiente antes dos demais e então cria um mapa através do SLAM. Os outros robôs usariam apenas uma câmera monocular e a partir do treinamento da rede, esses robôs menos complexos terão uma posição dentro do ambiente, sem necessitar de sensores mais complexos para a odometria. Também avaliamos nossa odometria através da comparação de medições reais com a medição do laser. O passo a passo dessa aplicação é descrito na Figura 1.4.

Figura 1.3 – Robô à direita explora o ambiente e cria o dataset para rede neural. Robô à esquerda explora o ambiente utilizando sua câmera monocular para cálculo de odometria.

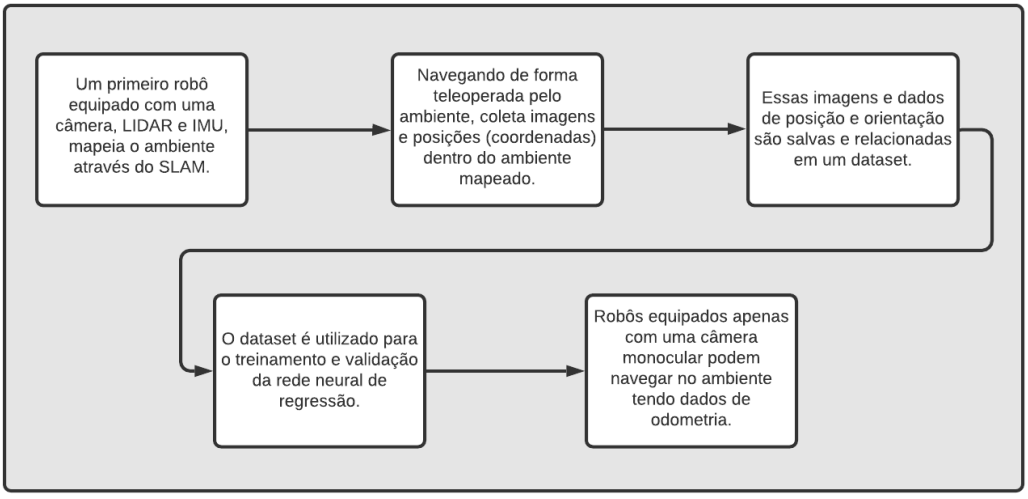


Figura 1.4 – Passo a passo de nossa metodologia de treinamento e inferência do modelo de aprendizado profundo para odometria visual.

1.1 Organização do Trabalho

Para descrever nossa abordagem, apresentamos a revisão bibliográfica no Capítulo 2, onde temos os referências teóricas e trabalhos relacionados. No Capítulo 3, descrevemos a plataforma robótica utilizada neste trabalho e as configurações dos algoritmos e técnicas utilizadas. Também apresentamos a metodologia de coleta dos dados para treinamento, o processo de treinamento e avaliação das redes neurais treinadas. No Capítulo 4 mostramos os resultados obtidos pela comparação das arquiteturas das redes neurais, comparação da arquitetura *ResNet* com o algoritmo da literatura Orb-Slam (MUR-ARTAL; MONTIEL; TARDÓS, 2015) e também uma avaliação de desempenho da arquitetura *ResNet* em sistemas embarcados. E por fim, no Capítulo 5.1 discutimos os resultados obtidos e trabalhos futuros.

1.1.1 Estrutura da Monografia

Capítulo 1: Introdução.

Capítulo 2: Revisão Bibliográfica/Embasamento Teórico.

Capítulo 3: Metodologia.

Capítulo 4: Resultados e Discussões.

Capítulo 5.1: Conclusão, Trabalhos Futuros e Publicações.

2 Revisão Bibliográfica

Neste capítulo, apresentamos o referencial teórico para a proposta de trabalho. Inicialmente, discutimos redes neurais e seu histórico, posteriormente abordamos a perspectiva da *Edge-AI* e suas definições, também discutimos algumas implementações. Em seguida, apresentamos discussões sobre robótica móvel e algoritmos de SLAM. Esses são os conceitos-chave por trás da ideia proposta neste trabalho. Posteriormente, discutimos os trabalhos relacionados à aplicação descrita neste trabalho.

2.1 Fundamentação Teórica

Como mencionado anteriormente, a fundamentação teórica deste trabalho se divide em três conceitos fundamentais: redes neurais, *edge-AI* e robótica móvel. Aqui, apresentamos os principais aspectos considerados para os temas de pesquisa citados.

2.1.1 Redes Neurais

Neste trabalho, utilizamos conceitos de redes neurais profundas e aprendizado profundo. As redes neurais profundas são um conjunto de unidades de ativação, também conhecidas como perceptron ou nós, que tem como objetivo julgar o resultado de uma operação matemática a partir de pesos pré-definidos. Estruturalmente, as redes neurais são distribuídas em unidades (nós ou perceptrons) e arestas (cada uma tem um peso atribuído) que uma vez interconectados formam as redes neurais. A primeira rede neural foi introduzida por McCulloch e Pitts, em 1943 no trabalho "A logical calculus of the ideas immanent in nervous activity"(MCCULLOCH; PITTS, 1943), onde os autores discorrem sobre como neurônios devem funcionar e modelam sua rede neural através de circuitos eletrônicos simples. Nos trabalhos de McCulloch e Pitts, as redes neurais só possuiam uma camada, foi então que anos depois, Kunihiko Fukushima introduziu as redes neurais multicamadas em 1975 (FUKUSHIMA, 1975). Uma camada de uma rede neural é um conjunto de nós. As redes neurais multicamada são o conceito chave para o aprendizado profundo, pois as redes neurais profundas são classificadas dessa forma pois possuem várias camadas. Uma rede neural profunda possui várias camadas e a maioria delas são denominadas de camadas ocultas, que são as camadas mais internas de sua composição. A Figura 2.1 demonstra uma rede neural profunda.

O funcionamento de uma rede neural, se baseia em que um nó se comporta como um neurônio humano. Logo, os nós são ativados quando há estímulos ou entradas suficientes. O resultado dessa operação se espalha através da rede, criando um resultado como resposta ao estímulo inicial. As arestas que conectam os nós atuam como sinapses simples, o que faz com que

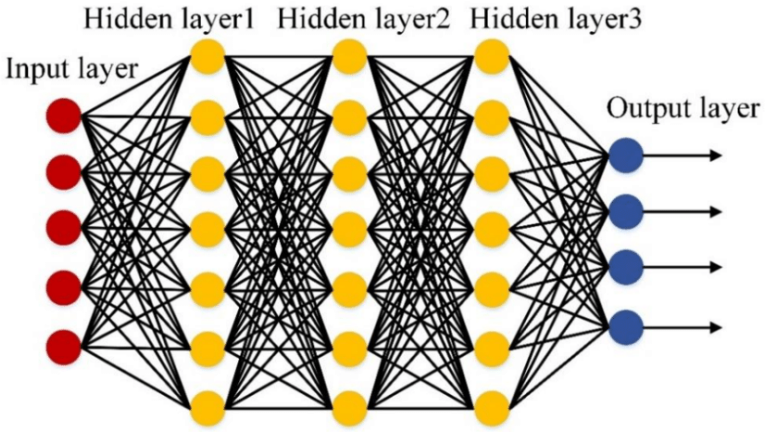


Figura 2.1 – Estrutura de uma rede neural profunda. Em vermelho temos os nós de de entrada da rede, ou a camada de entrada. Em amarelo temos as três camadas ocultas. E por fim, em azul a camada de saída. Todas as conexões entre os nós e camadas (em preto) são as arestas, em que cada uma tem um peso definido. (Imagem pertencente a Nosratabadi et al. (NOSRATABADI et al., 2020))

os sinais ou resultados passem de um nó para o outro. Então o estímulo inicial é passado até o final da rede com o objetivo de se ter um resultado, que geralmente é uma classificação. O processo de aprendizado das redes neurais consiste no treinamento, onde os dados de treinamento são alimentados na rede e assim seus pesos são definidos. Nesse processo de treinamento é necessário a coleta dos mais variados dados que representam o problema de predição ou reconhecimento a ser resolvido pela rede neural. Então é necessário coletar esses dados e os agrupar em um conjunto de dados ou *dataset*. Esse *dataset* será utilizado como a verdade absoluta para a rede neural, e é chamado de *ground truth* que em uma tradução livre, se traduz como uma "verdade essencial"ou "verdade base"da rede neural. O *ground truth* é necessário para o aprendizado supervisado, pois a rede pode aprender através da verificação do *ground truth*. Logo, esse processo de aprendizado ajusta os pesos de suas arestas a fim de se obter um resultado mais próximo possível do *ground truth* de treinamento.

Inicialmente se tinha dificuldade em criar métodos de aprendizado para redes neurais profundas. As primeiras abordagens, como a *ImageNet* (KRIZHEVSKY; SUTSKEVER; HINTON, 2012) e VGG (SIMONYAN; ZISSERMAN, 2014) possuíam poucos números de camadas, como por exemplo a *VGG* tem suas versões de 16 e 19 camadas. Teoricamente, um número maior de camadas resultaria em um menor erro de classificação, no entanto, o que acontecia na prática era um aumento do erro para números maiores de camadas como 34 camadas. Pensando em resolver esse problema, He et al. (HE et al., 2016) introduziu as *ResNets* que tem esse nome pois são redes residuais ou *Residual Networks*. A partir das *ResNets*, foi possível diminuir os erros para um número de camadas tão grande quanto 102 camadas. Neste trabalho, aplicamos as *ResNets* para a resolução do problema de odometria visual e comparamos à arquitetura *VGG*. No Capítulo 4 apresentamos esses testes e explicamos mais a fundo porque as *ResNets* funcionam melhor com mais camadas.

Através das CNNs (Convolutional Neural Networks), que são redes neurais de convolução como as *ResNets, VGG* e *ImageNet*, vários problemas de detecção, segmentação e classificação em imagens foram resolvidos, como mostrado em Sousa et al. (SOUSA et al., 2021). A convolução é uma operação matemática em que nas redes neurais tem uma aplicação de detecção de bordas e padrões nas imagens. Uma convolução é aplicada como uma operação das camadas das redes neurais. Como nosso problema é diretamente ligado à processamento de imagens, utilizamos as redes neurais para previsão de uma posição baseada em uma imagem. Essa metodologia visa relacionar uma imagem à uma coordenada no ambiente conhecido, além disso treinamos uma rede neural convolucional com o intuito de aprender as características de uma posição do ambiente através da imagem correspondente à aquela coordenada.

Em redes neurais também temos os chamados *backbones*, que são arquiteturas de redes neurais que são comumente utilizados em CNNs. As arquiteturas previamente citadas, como *ResNet* e *VGG* são exemplos de *backbones* e são utilizadas como tal pois suas arquiteturas são aproveitadas para gerarem outros tipos de resultados. Logo, se adiciona um *backbone ResNet*, com suas camadas convolucionais e logo após as camadas do *backbone* se adiciona mais camadas para obter um resultado desejado. Neste trabalho, utilizamos os backbones *ResNet* e *VGG* a fim de comparar a convergência desses *backbones* em nosso conjunto de dados para treinamento.

2.1.2 Inteligência Artificial na Borda - Edge-AI

Inicialmente abordamos as diferenças entre aplicações em *edge* e *cloud*. As aplicações em *cloud* são definidas assim pois todo, ou a maior parte do processamento computacional é executado fora do dispositivo que recolhe ou mostra os dados. Logo, todos os dados coletados ou a serem mostrados devem ser enviados por rede para serem processados ou esses dados são processados em algum servidor que está fisicamente distante (representando a *cloud* (nuvem) do nome). As aplicações em *edge* (borda), por sua vez, devem processar todos os dados diretamente em seus dispositivos de hardware com nenhuma ou rara comunicação com servidores ou dispositivos externos. Logo, o nome *edge* está diretamente conectado com o fato de todo processamento ser feito na borda da aplicação, ou seja, o mais próximo possível de onde o resultado é requisitado. O termo *edge-ai* deriva da necessidade de se processar aplicações de inteligência artificial na borda. Neste trabalho, apresentamos uma abordagem em *edge-ai* pois treinamos um modelo de rede neural profunda que precisa ser executado na borda da aplicação, ou seja, nos hardwares diretamente embarcados no robô.

Segundo Wang et al. (WANG et al., 2020), a era da informação, impulsionada pelo aumento significativo em dispositivos de computação e armazenamento está levando os serviços de computação em nuvem ao limite. As aplicações de inteligência artificial (IA) através dos conceitos de redes neurais profundas (DNNs) destacam os desafios de se realizar tarefas intensivas na nuvem e com recursos computacionais limitados. Atualmente, os benefícios da inteligência artificial na borda (*edge-ai*) também podem ser introduzidos na Internet das Coisas (*IoT*), com

dispositivos embarcados em *edge* mais poderosos computacionalmente (LI et al., 2019; LIN et al., 2019; SHI et al., 2020).

Os fatores que limitam as implementações de DL (Deep Learning - aprendizado profundo) em serviços de nuvem inteligente são:

- Custos de treinamento e inferência, que transmitem grandes volumes de dados para a nuvem e grandes volumes de tarefas a serem processadas, que então sofrem com as restrições de capacidade da rede.
- Potência da infraestrutura de computação e requisitos de latência mais rígidos para algumas aplicações.
- Congestionamento de tráfego devido ao alto volume de dados.
- Confiabilidade, que deve ser mantida mesmo quando as conexões de rede são perdidas.
- Privacidade, pois os aplicativos DL podem conter informações privadas.
- Alto consumo de energia;
- Baixo desempenho em tempo real e baixa qualidade de experiência do usuário (WANG et al., 2020; LI et al., 2019; SHI et al., 2020).

A implementação de soluções de IA na borda tem benefícios reconhecidos, mas ainda é considerado um problema em aberto. A inferência de soluções é sensível em termos de velocidade e precisão, já que dispositivos embarcados na borda são geralmente equipados com recursos computacionais e de armazenamento limitados (CHOI; JUNG, 2021; SHARMA et al., 2018). Além disso, a compressão de modelos de DL para implantação em dispositivos de borda pode perder desempenho e precisão, como é apresentado por Sharma et al. (SHARMA et al., 2018).

Em Shi et al. (SHI et al., 2020), são apresentados os principais desafios para a comunicação de sistemas de IA na borda. Além disso, o trabalho apresenta técnicas eficientes para tarefas de treinamento e inferência em redes neurais de aprendizado profundo. Segundo Lin et al. (LIN et al., 2020), o alto crescimento da computação móvel conectada à internet e aplicações em IoT gerará zilhões de bytes de dados na borda. Esse fato, consequentemente ocasionará em um grande volume de processamento de IA na borda, o que empurraria os limites atuais de hardware de dispositivos embarcados na borda.

Vários trabalhos também investigam implementações e otimizações de aplicativos de IA que conciliam DNNs tanto na borda quanto na nuvem. Choi e Jung (CHOI; JUNG, 2021) realizaram experimentos no *RefineDet* que se trata de uma arquitetura de detecção de objetos. Seus experimentos são baseados no uso várias arquiteturas de *backbone* em três plataformas diferentes: em um sistema desktop com uma GPU NVIDIA Titan XP, nas placas embarcadas

Drive PX2 e Jetson Xavier. Os resultados indicaram um desempenho de inferência em tempo real de aproximadamente 20 fps nas plataformas embarcadas.

Mazzia et al. (MAZZIA et al., 2020) implementa uma solução de IA em borda, que necessita de conceitos de processamento em tempo real para detecção de maçãs em plantações. Este trabalho, é baseado em uma arquitetura *YOLOv3-tiny* e é avaliada, de acordo com seu tempo de inferência, em três plataformas embarcadas. Klippel et al. (KLIPPEL et al., 2020) implementa uma solução de IA de borda para detectar falhas em correias transportadoras de minério de ferro. Logo, podemos concluir que trabalhos direcionados à aplicações de IA na borda contribuem cada vez mais para a implantação de soluções inteligentes em situações onde a conexão de internet pode ser limitada e nuvem não pode ser acionada.

Todas as soluções mencionadas reforçam a importância da *edge-ai* em aplicações na borda com baixo consumo energético. Em nossa abordagem, obter modelos de DL eficientes passa pela validação de uma arquitetura que convirja com o conjunto de dados e que ao mesmo tempo, seu processo de inferência seja computacionalmente permissivo. Logo, devemos não só validar uma arquitetura que tenha bons resultados, mas que não seja complexa computacionalmente pois elas deverão ser executadas em hardwares embarcados de baixo consumo energético. Pensando nisso, nosso trabalho não só avalia a convergência das arquiteturas de redes neurais propostas, como o desempenho das mesmas em plataformas embarcadas. Esses testes são apresentados no Capítulo 4.

2.1.3 Robótica Móvel e SLAM

Robôs móveis são dispositivos mecânicos que são capazes de atuar, interagir e se locomover um ambiente de navegação. Logo, para realizar as tarefas mencionadas, é necessário que toda a comunicação entre sensores, atuadores e algoritmos esteja regulada. Para realizar esta tarefa de regulação de dados, uma série de sensores, atuadores e um tipo de inteligência (por meio de algoritmos ou rotinas) devem ser implementados no robô (JAULIN, 2019).

O ROS (Robot Operating System) é um meta sistema operacional de código aberto para robótica. O ROS possui variados serviços e recursos comumente presentes em sistemas operacionais, que são direcionados a regulamentar a comunicação entre sensores, atuadores e algoritmos utilizados nos robôs construídos através do mesmo. Essas características podem facilitar o processo de construção de um robô modular e coeso devido à sua arquitetura de *publisher/subscriber* (QUIGLEY et al., 2009). Podemos destacar também, à ferramenta (presente no ROS) direcionada à visualização 3D e interação com as detecções do ambiente de navegação do robô, chamada *RViz*, como é mostrado no trabalho de Cid et al. (CID et al., 2020). O *RViz* ajuda o desenvolvedor a entender como o robô está interpretando os dados detectados, o que também é útil para depuração dos códigos, sensores e ligações elétricas. Outra vantagem do ROS é a ampla disponibilidade de algoritmos de código aberto, desde controladores de motores simples até técnicas SLAM mais complexas. Assim, usamos ROS em nosso robô principal para

utilização de algumas de suas técnicas públicas de SLAM a fim de coletar o *ground truth* da rede neural.

No ambiente da comunidade do ROS, existem diferentes algoritmos SLAM disponíveis para uso. Cada um desses algoritmos, pode ser utilizado direcionado a um caso de uso onde que combina com os sensores presentes no robô. Neste trabalho, usamos um laser planar para o processo SLAM. Em Santos et al. (SANTOS; PORTUGAL; ROCHA, 2013) observamos uma avaliação de algoritmos comuns de SLAM (presentes no ROS) que utilizam sensores de laser planar como sua principal fonte de dados. De acordo com Santos et al. os algoritmos de SLAM *gmapping* (GRISETTI; STACHNISS; BURGARD, 2007) e *HectorSlam* (KOHLBRECHER et al., 2011) mostram um dos menores acúmulos de erros médios (entre os algoritmos listados no trabalho) para a tarefa de SLAM utilizando lasers planares 2D. O artigo também mostra um dos problemas mais comuns no SLAM baseado em laser, que é a odometria. A odometria correta é crucial para a exatidão do mapa gerado no processo SLAM. No Capítulo 3 apresentamos os resultados da aplicação de ambos os algoritmos em nosso robô.

Cada vez mais, robôs móveis precisam de um maior poder de processamento em hardware embarcado. Isso se dá devido as crescentes características sensoriais e de inteligência das tarefas executadas por robôs que por sua vez estão cada vez mais complexas. Logo, podemos observar duas abordagens viáveis, são elas: usar um hardware embarcado mais poderoso computacionalmente ou distribuir a computação através da rede entre outros dispositivos computacionais. Devido à natureza de sua arquitetura *publisher/subscriber*, o ROS permite uma maneira fácil e padronizada de se distribuir a computação em vários dispositivos computacionais através da (MUNERA et al., 2017) (LIU; NIU; WANG, 2017).

Em Silva et al. (SILVA et al., 2020), é introduzido um teste de qualidade dos dados trocados entre um número de dispositivos computacionais que consomem dados produzidos por um robô móvel. Neste trabalho, é mostrado que o fator de qualidade dos dados em tempo real, diminui à medida que mais dispositivos se conectam a mesma rede. logo, podemos concluir que, em casos de uso onde o processamento em tempo real é uma demanda e além disso, também dependem de dispositivos conectados ao mesmo ponto de rede, se torna interessante ter dispositivos computacionais embarcados mais poderosas incorporadas ao robô. Esta análise também descreve o paradigma de computação na borda (SHI; DUSTDAR, 2016), já abordado na seção anterior.

Khan et al. concluem que os dispositivos de computação de borda visam trazer serviços e recursos da computação em nuvem para mais perto da aplicação, e assim garantir o processamento rápido das informações (KHAN et al., 2019). O trabalho deles também menciona que um dos desafios em aberto na computação de borda é o gerenciamento de recursos, que é um aspecto que consideramos em nosso trabalho. O processo de inferir informações em redes neurais é uma tarefa computacional intensiva e que pode consumir uma quantidade considerável de energia.

O ROS tem um papel de destaque na literatura de robótica móvel, o meta sistema operacio-

nal revolucionou as aplicações por não só modularizar a construção de soluções, mas também por sua cultura de código de aberto. Neste trabalho, utilizamos múltiplas soluções de código aberto que estão disponíveis dentro do ROS, como os algoritmos de SLAM mencionados *GMapping e HectorSlam*, além de outros códigos de menor complexidade como controladores de motores e sensores que são disponibilizados por fabricantes [1] ou pela comunidade em geral [2].

2.2 Trabalhos Relacionados

Como mencionado anteriormente, o cálculo de odometria é crítico em dispositivos robóticos, além disso, existem diferentes técnicas e sensores associados à sua medição. Nossos trabalhos relacionados se concentram em trabalhos que utilizam de alguma forma o aprendizado profundo, lasers e odometria visual em plataformas de robótica móvel. Posteriormente, destacamos as diferenças entre nossa abordagem para os outros trabalhos.

Em Shan et al. (SHAN; ENGLOT, 2018) observamos uma técnia para cálculo de odometria menos complexa computacionalmente e baseada em lasers espaciais (3D) e otimizada para o movimento considerando o solo. Esse trabalho também se preocupa em implementar um algoritmo menos complexo computacionalmente. Isso se dá pelo objetivo de se ter um melhor desempenho em dispositivos embarcados, como os produtos da família Jetson da NVIDIA, onde sua abordagem também é testada e avaliada. A parte principal dessa abordagem está na medição de odometria que depende da detecção do solo, pelo LIDAR. Isso torna o caso de uso dessa estratégia restrito à lasers espaciais 3D, o que pode aumentar consideravelmente o custo da plataforma robótica. Em comparação com nossa abordagem, nos concentramos em ter apenas um robô com um laser planar, que tem um baixo custo comparado a lasers espaciais que tem como objetivo mapear o ambiente de navegação. Em seguida, o mapa criado pela técnica de SLAM alimenta os dados de treinamento de uma rede neural, o que permite que qualquer outro robô percorra o ambiente de navegação com apenas uma câmera monocular.

Em Zhan et al. (ZHAN et al., 2020) e Li et al. (LI et al., 2018), ambos propõe um cálculo de odometria através de câmeras monoculares. As abordagens calculam a odometria por meio da predição da movimentação dos pixels nas imagens monoculares. Essa abordagem pode ser observada na Figura 2.2. Essa técnica de predição, é denominada como *optical-flow* e permite a aferição de dados de profundidade presentes nas imagens, o que melhora a compreensão de movimento entre imagens. Ambas as abordagens também usam aprendizado profundo para prever a distância do movimento do pixel entre os quadros no tempo 1 e no tempo 2. Os dados de movimentação permitem que a rede proposta extraia informações de profundidade nas imagens monoculares e preveja a posição e a orientação do veículo. O trabalho também destaca o desempenho alcançado através de testes no *dataset* KITTI, no entanto, os resultados dos dois trabalhos não podem ser comparados entre si devido às diferentes métricas usadas para medir os erros de predição das

[1] https://github.com/YDLIDAR/ydlidar_ros
[2] https://github.com/isarlab-department-engineering/ros-motor-hat

posições. Enquanto Zhan et al. usa um erro de posição e orientação relativo à diferença de quadro a quadro, Li et al. usa uma função de erro quadrático médio para calcular o erro de deslize (*drift*). Em nossa abordagem, não nos preocupamos em medir as variações de pixels entre imagens, apenas relacionamos uma imagem com uma posição no ambiente de navegação mapeado. Em seguida, essas posições com suas imagens relativas são alimentadas em um treinamento de uma rede neural de regressão que tem objetivo de prever a posição do robô móvel dentro do ambiente de navegação previamente mapeado pelo algoritmo de SLAM.

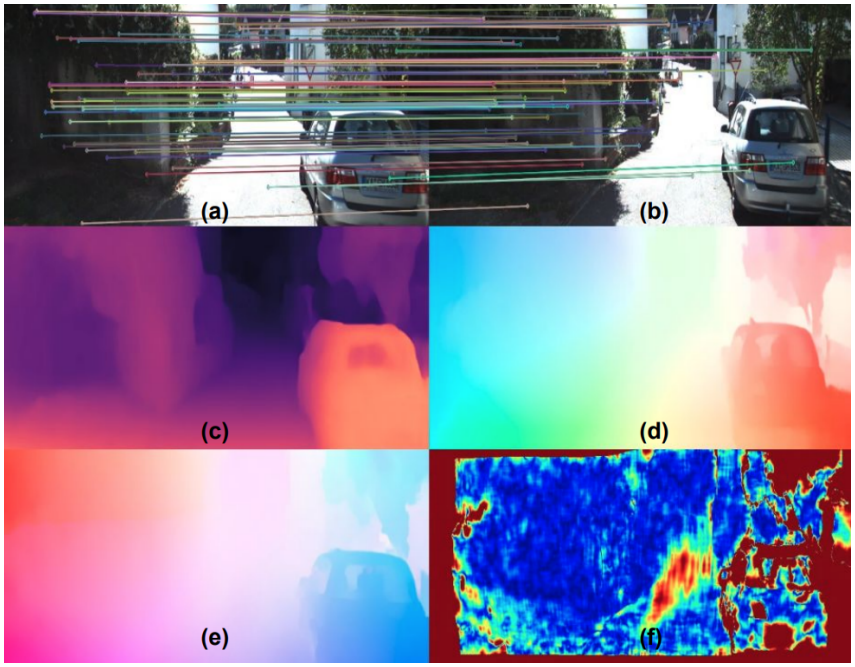


Figura 2.2 – Figura presente em Zhan et al. (ZHAN et al., 2020), onde o processo de inferência de profundidade em imagens monoculares é descrito. Em **(a)** e **(b)** temos os dois frames t1 e t2 com exemplos de combinações de padrão 2D. Em **(c)** a predição de profundidade. Em **(d)** e **(e)** a predição de *forward e backward optical-flow*. E por fim, em **(f)** a consistência do fluxo de pixels usando as predições em **(d)** e **(e)**.

Na abordagem descrita por Yan et al. (YAN et al., 2017), dois algoritmos de odometria visual são fundidos a fim de um cobrir os defeitos do outro. Os dois algoritmos são o algoritmo LOAM (Lidar odometry and mapping) (ZHANG; SINGH, 2014) e o algoritmo viso2 (GEIGER; ZIEGLER; STILLER, 2011). Os dados de saída dos dois algoritmos são fundidos a fim de extrair informações de odometria. O algoritmo viso2 possui uma taxa de erro conhecida ao se utilizar câmeras monoculares (também possui uma versão mais acurada utilizando câmeras stereo). O trabalho propõe a avaliação de três métodos de fusão dos dados para se tirar proveito das maiores vantagens de ambos algoritmos. As métricas visam a melhora da precisão e diminuição do erro de *drift* do cálculo odométrico. Em comparação à nossa abordagem, também fundimos o dados de odometria com dados de orientação, através do pacote *rf2o* (disponível no ROS) e as informações de leitura de orientação da IMU. Esse método de fusão visa a construção do conjunto de dados utilizado como *ground truth* do treinamento da rede neural de detecção da posição do robô no

ambiente.

O algoritmo Orb-Slam, descrito na abordagem de Mur-Artal et al. (MUR-ARTAL; MONTIEL; TARDÓS, 2015), é um algoritmo de SLAM visual. Assim como na odometria visual, os algoritmos de SLAM visual também utilizam câmeras stereo e monoculares. O Orb-Slam não é exclusivamente um algoritmo de odometria, no entanto, utiliza de cálculos odométricos para estimar a posição do robô no ambiente. Esse cálculo se faz necessário para a parte de localização do SLAM. Em sua abordagem, os autores se baseiam no ORB (RUBLEE et al., 2011), que se trata de um descritor binário de alta performance. O ORB relaciona pontos em duas imagens distintas porém relacionadas, em que a sua diferença é baseada na variação de suas posições e orientações. O algoritmo faz essa relação através da fixação de pontos de interesse na imagem *i1* e tenta achar esses mesmos pontos na imagem *i2*, a Figura 2.3 demonstra esse processo. Esse processo pode ser útil em casos onde é necessário estimar o movimento entre duas imagens com intuito de estimar a orientação e a posição de um robô no ambiente. O algoritmo Orb-Slam é comparado à nossa abordagem e descrito no Capítulo 4.

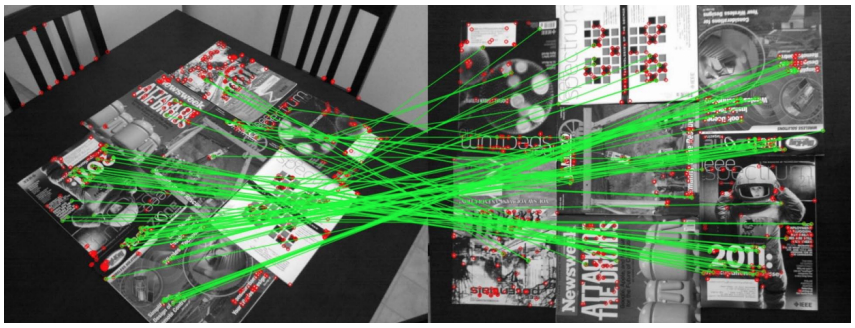


Figura 2.3 – Figura presente em Rublee et al. (RUBLEE et al., 2011), que mostra o processo de rastreamento entre duas imagens.

3 Desenvolvimento

Neste capítulo, apresentamos a plataforma robótica desenvolvida e a metodologia de treinamento da rede neural para odometria visual. O texto se divide em três seções, uma que descreve a plataforma robótica e suas propriedades, outra seção que apresenta a metodologia de construção do conjunto de dados (*ground truth* de treinamento da rede neural) e a última seção detalha a metodologia de treinamento das arquiteturas de redes neurais.

3.1 Plataforma Robótica

A plataforma robótica desenvolvida neste trabalho, utiliza o hardware Raspberry Pi 4 como sua unidade principal de processamento. Esta unidade representa o nó mestre de acordo com a arquitetura do ROS. O nó mestre se baseia na unidade onde a maioria dos algoritmos são executados. A Raspberry Pi é um computador de arquitetura ARM que tem um tamanho físico próximo a de um cartã ode crédito. O seu tamanho é benéfico para aplicações móveis onde o espaço é limitado. Como o chassi de nosso robô tem medidas em torno de 30x10x15 centímetros (comprimento, largura, altura), ela se torna uma opção viável. Além disso a Raspberry Pi 4, conta com 4GB de RAM e uma CPU ARM Cortex-A72 Quad Core. A Figura 3.1 demonstra o sistema embarcado.

Figura 3.1 – Hardware embarcado Raspberry Pi 4 Model B.[1]

O robô é equipado com quatro motores de 12V de corrente contínua conectados a rodas suecas, essas rodas habilitam uma locomoção omnidirecional no ambiente. A locomoção omnidimensional, através das rodas suecas, permite o robô se movimentar em múltiplos eixos, como mostrado na Figura 3.2. O robô também conta com uma IMU conectada via I2C a um Arduino Uno, que se comunica com o nó mestre (Raspberry) através de portas seriais (USB). Sensores de IMU são úteis para aferição da orientação do robô no ambiente. Nosso principal do

[1] Imagem retirada de https://pt.wikipedia.org/wiki/Raspberry_Pi. Acesso em 13/12/21.

sensor para o processo de SLAM, se baseia no laser planar LIDAR X2 da Ydlidar. O LIDAR X2 é um laser com relativo baixo custo, considerando os padrões de custo de lasers LIDAR. O laser possui uma frequência de varredura de 7Hz e uma distância máxima de alcance de 8 metros, o que é considerado uma distância máxima suficiente para os propósitos de testes internos e domésticos. Usamos o script de processamento disponibilizado no ROS pela fabricante do laser para publicar as leituras no ambiente. Esse tipo de dado é publicado na comunicação do ROS como um tipo de mensagem nativa, chamada *LaserScan*. Os lasers são úteis pois suas leituras retornam as distâncias nos arredores do robô, no caso do laser utilizado, o obstáculo mais longe a ser identificado estará a 8 metros.

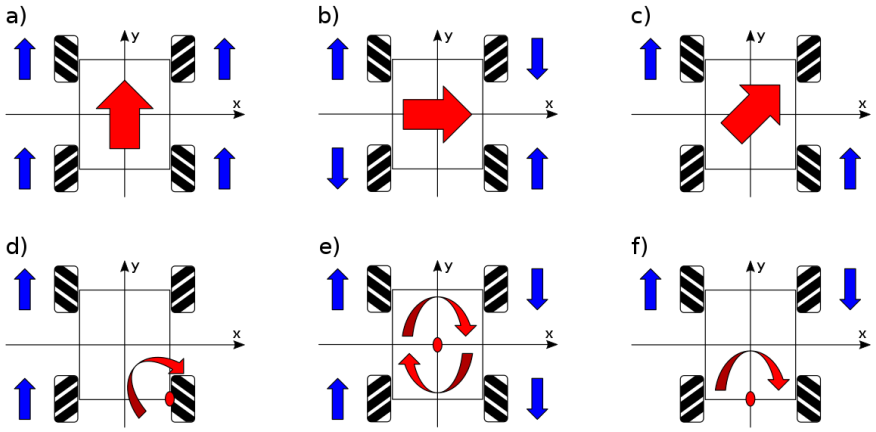


Figura 3.2 – Modelo de locomoção omnidirecional habilitado pelas rodas suecas.[2] As setas vermelhas representam a direção de locomoção e as setas azuis representam a direção de força das rodas para que o movimento seja realizado.

Em relação ao sistema de alimentação do robô, todos os quatro motores são alimentados por uma bateria Li-Po (polímeros de lítio) de 12V. O restante do sistema é alimentado por uma bateria simples de de 5V. Todos os sensores são alimentados diretamente pelas portas USB da Raspberry, inclusive o Arduino UNO que é utilizado em conjunto da IMU. Dessa forma todo o sistema energético está concentrado na Raspberry, exceto os motores que requerem uma maior tensão.

Também criamos um URDF (Universal Robot Description File) com o objetivo de melhor visualização da interação do robô com o mapa através da ferramenta *RViz*. O URDF também é necessário para exibir as transformações corretas da posição do robô no mapa SLAM. Os arquivos URDF são descrições precisas do robô, todas as medidas das rodas e do chassi são anotadas dentro desse arquivo para a criação das transformações adequadas. Esse arquivo também contém a árvore de TF's do robô, que se trata de um tipo de dado para as transformações dentro do ROS e são necessários para variados métodos presentes no meta sistema operacional. A Figura 3.3 mostra a plataforma robótica e seu respectivo URDF no *RViz* e a Figura 3.4 mostra as transformadas de cada parte do robô.

[2] Imagem retirada de https://www.wikiwand.com/en/Mecanum_wheel. Acesso em 13/12/21.

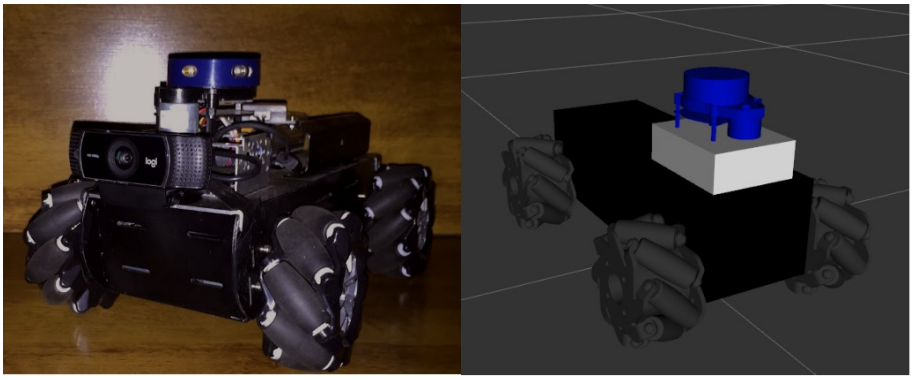


Figura 3.3 – Plataforma robótica real à direita e sua respectiva versão virtual descrita pelo URDF à esquerda.

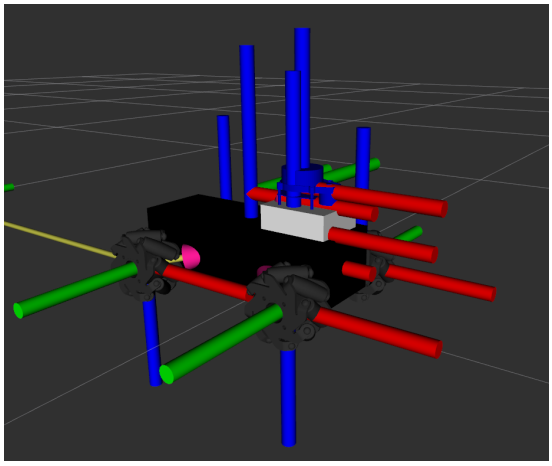


Figura 3.4 – Eixos das transformadas de cada parte do URDF do robô.

3.2 Metodologia de coleta de dados para o *ground truth*

Para coletar as coordenadas do ambiente de navegação utilizados como *ground truth* da rede neural, conforme mencionado no Capítulo 2, avaliamos dois métodos SLAM. Para o processo de mapeamento do ambiente através do SLAM, desenvolvemos um módulo de teleoperação para a coleta dos dados no ambiente. Devido às rodas suecas, o robô desenvolvido pode se mover livremente nos eixos de navegação terrestre (x, y), e por isso desenvolvemos um script de teleoperação omnidirecional.

Inicialmente, utilizamos o algoritmo *HectorSlam*, em sua versão onde não há necessidade de dados odométricos específicos. O *HectorSlam* possui um modo de configuração onde se obtém a odometria apenas através do laser e outros otimizadores globais. Este modo de configuração é útil em robôs ou dispositivos móveis, como drones, que não possuem um modelo claro para cálculos de odometria mais confiáveis. Com essa configuração, conseguimos obter dados sólidos de mapeamento, no entanto, a localização e orientação apenas através do laser resultou em alguns erros de mapeamento quando se fez necessário manobrar o robô para que o mesmo girasse em torno de seu próprio eixo de rotação (eixo Z). Nesse processo, observamos que o modelo

omnidireciona, habilitado pelas rodas suecas, foi útil para obter um bom mapa, visto que não era recomendável a rotação em seu próprio eixo. No entanto, visualmente, o mapa gerado ainda não era preciso o suficiente para ser usado como o *ground truth* de treinamento de uma rede neural. A Figura 3.5 mostra os erros obtidos ao se utilizar a versão de do algoritmo *HectorSlam* com dados de localização e orientação apenas por laser.

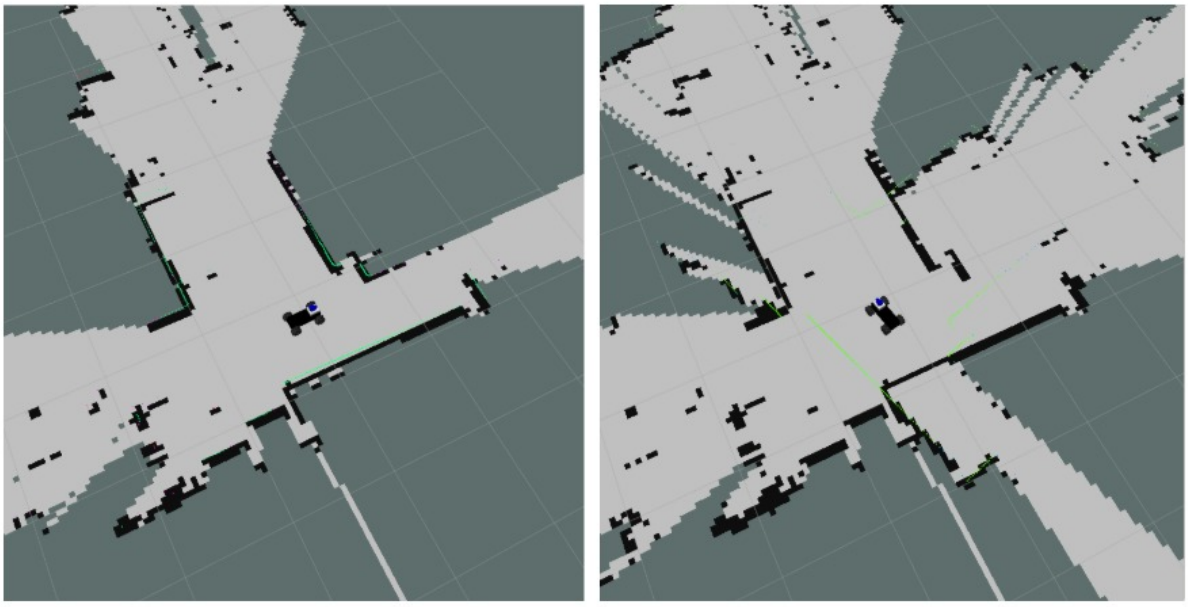


Figura 3.5 – Erros encontrados na utilização do algoritmo de SLAM *HectorSlam*.

Para melhorar o mapa gerado pelo alforitmo de SLAM, configuramos a odometria através do pacote ROS *rf2o* (JAIMEZ; MONROY; GONZÁLEZ-JIMÉNEZ, 2016). A odometria pelo *rf2o* também é realizada pelo laser planar, no entanto, é baseada em diferentes métodos com relação ao método utilizado no algoritmo *HectorSlam*. Este pacote mostra a posição do robô nos três eixos possíveis (x, y, z), onde os valores de saída são obtidos em metros. Para obter dados mais confiáveis de odometria, e potencialmente eliminar o erro de giro no próprio eixo, adicionamos uma IMU e fundimos as suas leituras de orientação com as informações de posição do laser, fornecidas pelo pacote *rf2o*. Os dados fundidos, são publicados no tópico */odom*, que é o tópicp padrão de odometria do ROS. A partir dos dados de localização providos pelo pacote *rf20*, validamos as posições de leitura do laser com uma fita métrica. A figura 3.6 ilustra este processo de validação.

O dados de saída do pacote *rf2o* fundidos com os dados de orientação advindos da IMU, obtiveram melhores resultados de mapeamento no *HectorSlam*, mesmo se robô gira em seu próprio eixo. No entanto, se o robô girasse de uma forma mais rápida, o mapa ainda perdia precisão. Logo, se fez necessário o teste de outros algoritmos de SLAM para verificarmos se os problemas estavam relacionados aos sensores ou ao algoritmo utilizado. Então, mudamos a abordagem para o algoritmo de SLAM *Gmapping*. Os resultados obtidos pelo *Gmapping* foram melhores, considerando o problema de rotação do robô em seu próprio eixo, logo, mudamos o

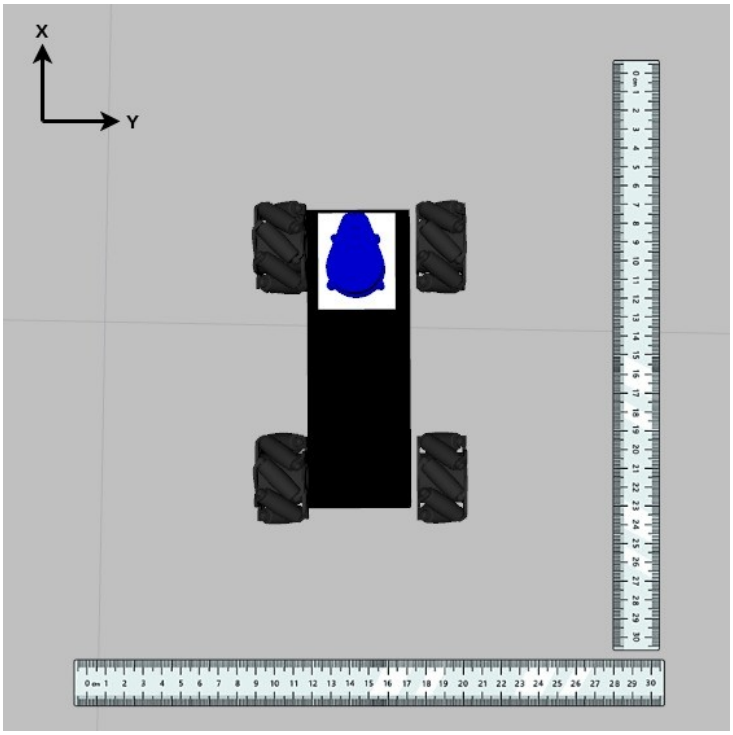


Figura 3.6 – Validação da odometria obtida pelo pacote ROS RF2O.

algoritmo de SLAM utilizado para o *Gmapping*. A figura 3.7 mostra o mapa do ambiente, gerado pelo algoritmo *Gmapping*.

Assim que foi possível se obter um mapa consideravelmente preciso e livre de erros (através do algoritmo *Gmapping*), construímos um script para relacionar uma imagem a uma posição específica dentro do mapa. Essa metodologia se baseia na teleoperação do robô no ambiente de navegação e então captura uma imagem por meio de uma câmera Logitech C922 USB e em seguida, recebe uma coordenada presente no tópico de odometria */odom* a fim de salvar a posição (coordenada x, y) e a orientação atuais do robô. O arquivo de anotação do *ground truth* de treinamento, consiste em um nome de arquivo de imagem, a posição atual x e y do robô, e também a orientação do robô calculada pela IMU através cálculo do valor do ângulo de rotação *yawn*. O ângulo de rotação *yawn* se trata da angulação de rotação em torno de seu próprio eixo, ou seja, o eixo Z.

Para criar o conjunto de dados, o robô foi teleoperado pelo ambiente de navegação e capturava as imagens e relacionava ás mesmas com uma determinada posição no mapa. Logo, fixamos um ponto de partida no ambiente de navegação a fim de ser a referência (origem, ponto zero) do nosso mapa. Assim, todas as vezes que dados foram coletados, foi necessário começar à coleta neste mesmo ponto fixado, para que então todos os dados de localização tivessem a mesma origem. Caso contrário, não fariam sentido pois poderiam confudir a rede neural em seu processo de treinamento. Neste primeiro momento, o conjunto de dados criado obteve 6203 imagens para treinamento e outras 1183 imagens para os procedimentos de teste e validação. O ambiente de

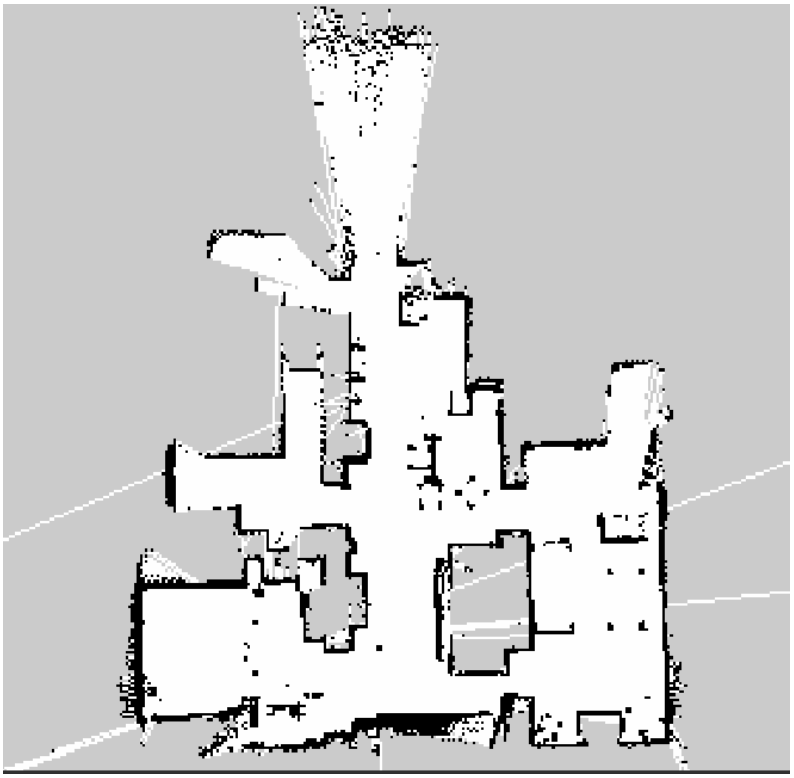


Figura 3.7 – Ambiente de coleta de dados para o *ground truth* mapeado pelo algoritmo *Gmapping*.

navegação do robô considerado para este trabalho é um cenário doméstico, onde o robô faz o mapeamento. A Figura 3.8 mostra exemplos das imagens coletadas no ambiente doméstico.

Figura 3.8 – Exemplo de imagens coletadas para o conjunto de dados no ambiente doméstico.

3.3 Processo de Treinamento

Para treinar o processo de treinamento da rede neural, avaliamos três arquiteturas de redes neurais. Todas as arquiteturas de rede avaliadas tem a mesma entrada (imagem RGB de 640x481) e as mesmas saídas (posição x e y no mapa e a orientação). As arquiteturas avaliadas são as seguintes: arquitetura de modelo customizada e outras duas com *backbones* conhecidos da literatura. Com relação ao modelo personalizado, rede de convolução simples foi construída, sendo ela cinco camadas convolucionais, alternando com cinco camadas de max-pooling 2D. Também adicionamos duas camadas densas no final para a normalização de dimensões das saídas. A função de ativação utilizada se baseia na ReLu para todas as camadas, exceto a última camada que tem uma função de ativação linear. A função de perda utilizada, se baseia na função de erro médio absoluto, e também utilizamos o Adam como otimizador. A segunda rede avaliada, é baseada no *backbone VGG-16* e a terceira no *backbone ResNet50*. No caso dos *backbones*, também adicionamos duas camadas densas ao final dos *backbones* a fim de gerar os valores desejados. A Figura 3.9 ilustra todas as arquiteturas que utilizadas neste trabalho. Também utilizamos os pesos iniciais da *ImageNet* para os modelos*ResNet* e VGG, o que beneficia o modelo para aprender mais rápido. Todos os modelos foram treinados por 250 épocas.

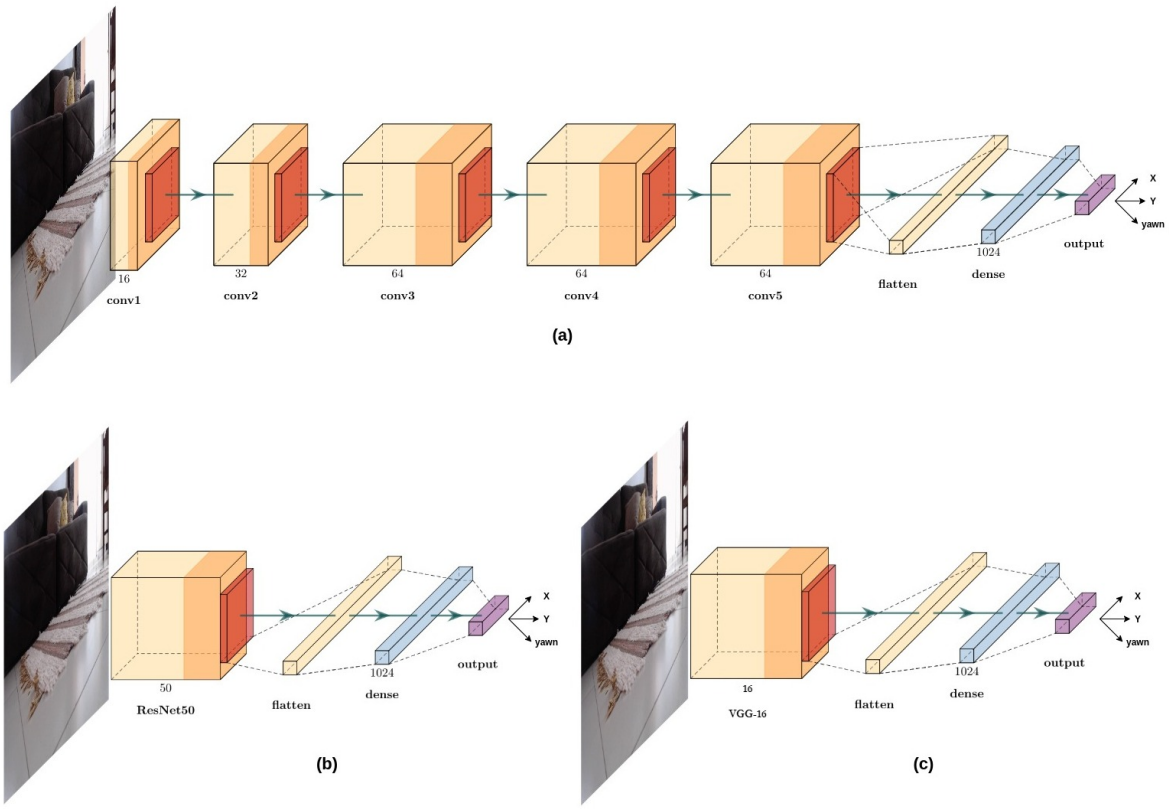


Figura 3.9 – Arquiteturas de redes neurais propostas. Em (a) o modelo customizado treinado, em (b) temos o backbone *ResNet50*, e em (c) o backbone *VGG-16*.

4 Resultados

Neste capítulo, apresentamos os resultados obtidos pela nossa metodologia de coleta do conjunto de dados e treinamento das redes neurais *VGG-16*, arquitetura customizada e *ResNet*. Também apresentamos testes de desempenho da rede neural *ResNet* em dispositivos embarcados, além de comparar o desempenho de nossa abordagem com o algoritmo da literatura Orb-Slam (MUR-ARTAL; MONTIEL; TARDÓS, 2015).

Para avaliar a precisão dos modelos treinados, utilizamos a distância euclidiana média como métrica, de acordo com a equação 4.1. Essa métrica calcula a distância entre p_i (coordenada (x,y) de previsão) e g_i (coordenada presente no *ground truth*). Em nossa abordagem, como utilizamos as coordenadas geradas pelo pacote *rf2o* como *ground truth*, as distâncias entre as previsões e o *ground truth* são dadas em metros. A distância média entre as previsões e o *ground truth*, além do desvio padrão do conjunto de distâncias encontrado em cada modelo, são mostrados na Tabela 4.1. Posteriormente, apresentamos um retreinamento do modelo *ResNet*, que obteve os melhores resultados de acordo com nossa métrica de avaliação. Também, comparamos a nossa solução com a versão monocular do algoritmo *Orb-Slam2* (MUR-ARTAL; MONTIEL; TARDÓS, 2015). E finalmente, mostramos testes de desempenho do modelo retreinado em plataformas embarcadas, utilizamos o número de quadros (*frames*) por segundo e relacionamos ao consumo de hardware nas respectivas plataformas.

$$D\left(p,g\right) = \sqrt{\sum_{i=1}^{n}\left(g_i - p_i\right)^2} \tag{4.1}$$

Modelo	Distância média (m)	Desvio Padrão
ResNet50	0.4587	0.4924
VGG-16	0.7587	0.5752
Modelo Customizado	0.6299	0.8774

Tabela 4.1 – Comparação de erro médio entre os modelos de redes neurais propostos.

A Tabela 4.1 mostra que o modelo com arquitetura *ResNet50* possui o menor erro médio e desvio padrão. Essa arquitetura obteve um erro médio de 45,87 centímetros, o que corresponde a cerca de 10,02% da maior distância em nosso ambiente de navegação, que é de 4,5758 metros. O modelo *VGG-16* apresentou o maior erro médio que correspondeu a 75,87 centímetros, tendo uma variação correspondente a 16,58% da nossa maior distância. Por fim, a arquitetura customizada apresentou erro médio de 62,99 centímetros, correspondendo a 13,76% da maior distância. Embora o modelo de arquitetura customizada tenha um erro médio menor do que o da arquitetura

VGG-16, seu desvio padrão foi consideravelmente maior, o que significa que o erro está mais disperso no conjunto de dados.

Para uma melhor avaliação dos modelos, também propomos uma forma visual de avaliação. Essa avaliação se dá por meio de um gráfico de dispersão. O gráfico de dispersão, mostra a correspondência das coordenadas presentes no *ground truth* com as previsões da rede neural. O gráfico de dispersão pode ser útil para a análise se os pontos de previsão previstos estão ou não se encaixando no *ground truth*, visto que usamos um modelo de regressão. A Figura 4.1 demonstra o gráfico de dispersão do modelo de arquitetura personalizada, onde podemos observar que ainda existem vários erros de previsão de posição do robô. Na Figura 4.2 é mostrado o gráfico de dispersão do modelo com *backboneVGG-16*, que indica que esse modelo também faz previsões inacuradas, visto que temos os pontos vermelhos do *ground truth* desalinhados com o os pontos azuis de predição. E por fim, na Figura 4.3, é mostrado o gráfico de dispersão do modelo com *backbone ResNet50*. Esse modelo foi o que melhor se ajustou ao *ground truth*. Os gráficos corroboram a nossa métrica de avaliação, exibida na Tabela 4.1. Assim, escolhemos o modelo *ResNet50* para uma avaliação mais profunda.

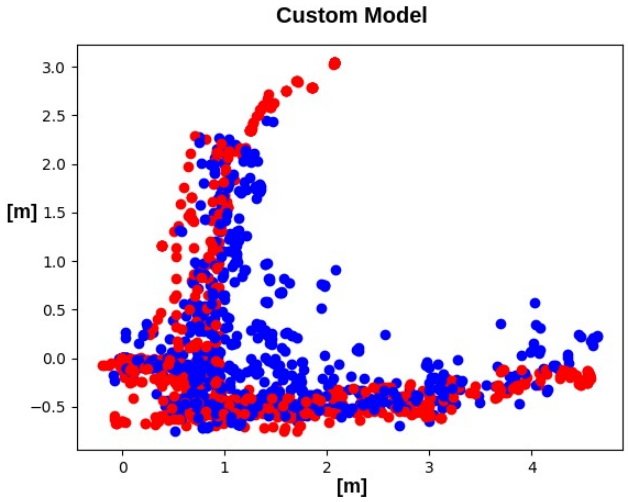


Figura 4.1 – Gráfico de dispersão da arquitetura customizada proposta. Pontos correspondentes ao *ground truth* em vermelho e pontos de predição em azul.

Para melhor avaliação do modelo *ResNet*, também analisamos o histograma de erro de predições desse modelo. O histograma visa agrupar dados correlatos em intervalos conhecidos da distribuição dos dados. Logo, relacionamos o valor do erro de cada amostra no eixo x e o número de amostras correspondentes aos erros no eixo y. A Figura 4.4 mostra o histograma de erro gerado por este modelo. Observamos na Figura 4.4 que a maioria dos erros está dentro da distância média de erro mostrada na Tabela 4.1. Também construímos um gráfico do tipo *boxplot* a fim de analisar a dispersão do erro no conjunto de dados. Esse tipo de gráfico particiona os dados em quatro quartis, onde esses quartis são definidos baseados na mediana dos erros. Esse gráfico também apresenta os *outliers* que são dados que estão fora do padrão do conjunto e são

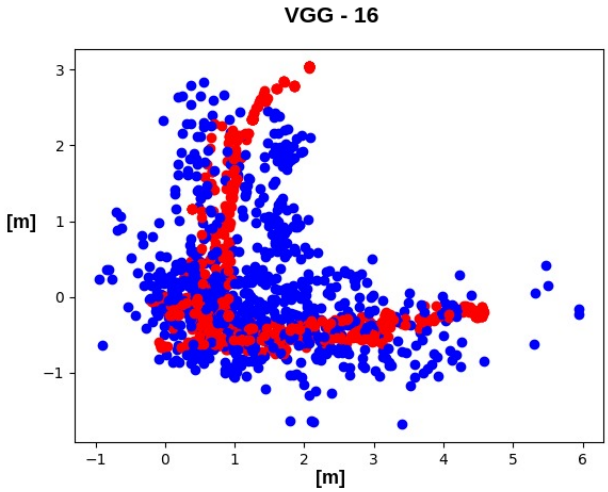


Figura 4.2 – Gráfico de dispersão do modelo com *backbone VGG-16*. Pontos correspondentes ao *ground truth* em vermelho e pontos de predição em azul.

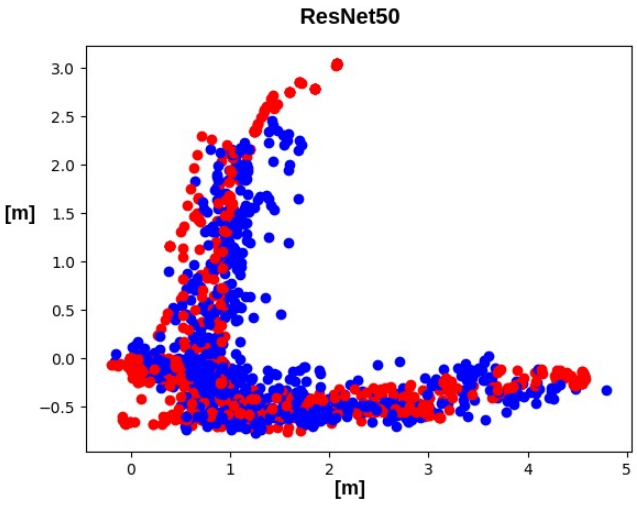


Figura 4.3 – Gráfico de dispersão do modelo com *backbone ResNet50*. Pontos correspondentes ao *ground truth* em vermelho e pontos de predição em azul.

representados acima do maior valor médio. Na Figura 4.5 podemos observar o gráfico do tipo *boxplot* correspondente ao modelo com *backbone ResNet*. Neste gráfico, podemos observar que o erro está distribuído de acordo com o erro médio mostrado na Tabela 4.1. Observamos também que o erro médio máximo está acima de 1 metro, no entanto, a maioria dos erros estão próximos à média apresentada na Tabela 4.1, fato que também é corroborado pelo histograma.

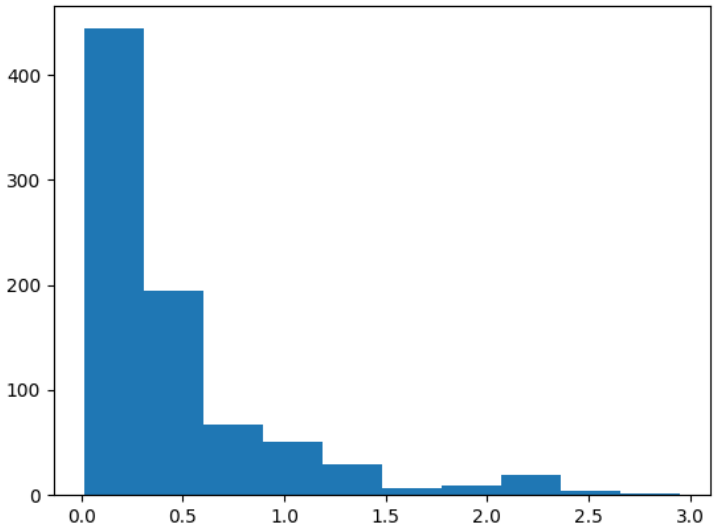


Figura 4.4 – Histograma de erros correspondente ao modelo com *ResNet50*. Eixo y corresponde ao número de amostras e o eixo x corresponde ao erro médio pertencente às amostras.

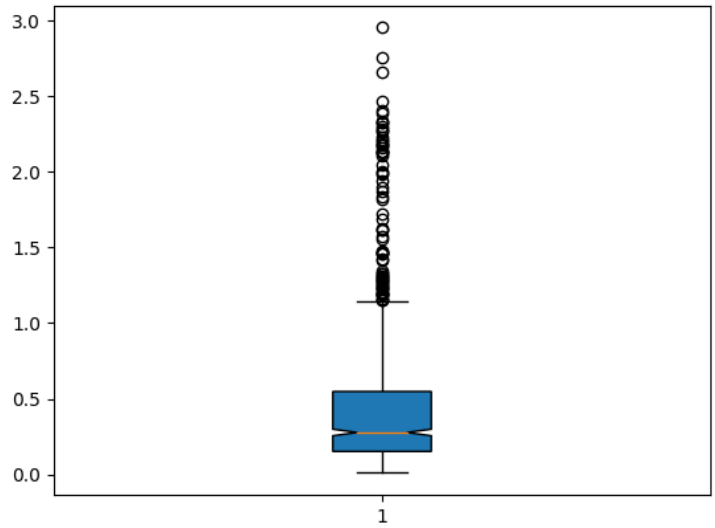


Figura 4.5 – Gráfico do tipo *boxplot* correspondente ao modelo *ResNet50*. A caixa azul corresponde ao intervalo da maioria dos valores no conjunto de dados. A linha amarela corresponde à mediana dos erros de predição. Os círculos pretos são os *outliers*.

Os resultados obtidos pelo modelo que utiliza a *ResNet* como backbone são melhores pois em sua arquitetura, apresentada por He (HE et al., 2016), é introduzido o conceito de blocos residuais. Esses blocos residuais podem melhorar o processo de aprendizado pois utilizam resultados obtidos pelas camadas de convolução anteriores a fim de não desperdiçar as informações de aprendizado. Logo, um conjunto de camadas de convolução se tornam um bloco residual, e vários desses blocos residuais formam uma *ResNet*. A Figura 4.6 demonstra um bloco residual, onde o valor x de uma camada anterior é somado ao resultado de uma função F(x) representada pelo processo interior das ativações e convoluções da camada atual.

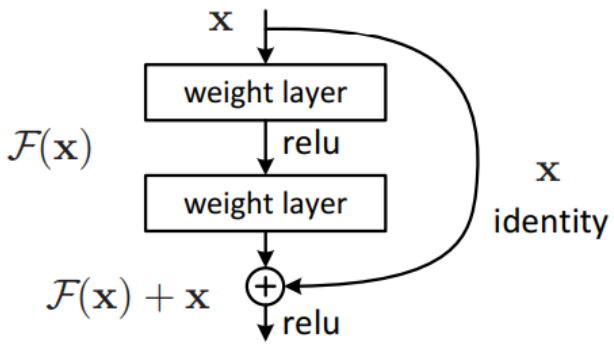


Figura 4.6 – Arquitetura do bloco residual, onde x é o resultado das operações anteriores e F(x) representa as funções aplicadas pela camada atual no valor de saída x. Figura encontrada em He (HE et al., 2016)

Outra contribuição relevante apresentada por He (HE et al., 2016) que corrobora os resultados obtidos é de que as *ResNets* diminuem o erro de acordo com maiores números de camadas. Em teoria, quanto maior a quantidade de camadas menor o erro, no entanto maiores números de camadas resultavam em problemas de perda ou explosão do gradiente. Logo, em arquiteturas anteriores a *ResNet* como a VGG, adicionar mais camadas resultava diretamente em maiores taxas de erro devido aos erros no gradiente. No entanto, os blocos residuais conseguiram diminuir essa taxa de erro através da soma de resultados anteriores, como mostrado na Figura 4.6.

4.1 Retreinamento do Modelo *ResNet50*

Após a validação do *backbone ResNet* como a arquitetura que melhor convergiu sobre o conjunto de dados, retreinamos nosso modelo de cálculo odométrico. O retreinamento consistiu no aumento do conjunto de dados que foi coletado anteriormente. No treinamento anterior, treinamos o modelo com 6203 imagens, enquanto nesse treinamento aumentamos para 8029 imagens com seus respectivos dados de posição no ambiente. Outra alteração com relação ao treinamento anterior é na resolução de entrada do modelo. Com o intuito de se ter uma menor complexidade ao ser interpretada, diminuímos a resolução de entrada do modelo para 320x240.

Essa resolução corresponde à metade da resolução de entrada anterior, que era de 640x480. Além dessas modificações, também mudamos os dados de entrada do dataset. Através da interface de carregamento de dados do *framework TensorFlow*, normalizamos os pixels das imagens do conjunto de dados de treinamento. Essa técnica de normalização visa ajustar os valores de intensidade dos pixels em uma média do total. Os resultados de erro médio de acordo com nossa métrica de distância euclidiana, são apresentados pela Tabela 4.2.

Modelo	Distância média (m)	Desvio Padrão
ResNet50 640x480	0.4587	0.4924
ResNet50 320x240	0.3519	0.3990

Tabela 4.2 – Comparação de erro médio entre os modelos com *backbone ResNet*.

Como podemos observar na Tabela 4.2 o retreinamento do modelo obteve uma melhora de 23,28% com relação ao erro médio. Apesar da diminuição da resolução de entrada, diminuímos o erro de forma considerável. Essa melhora ocorre principalmente pelo aumento dos dados de treinamento, o que indica que podemos ter um melhor modelo com mais dados. No entanto, não podemos ter essa certeza à priori, pois o comportamento de treinamento de modelos de aprendizado profundo ainda é de certa forma experimental.

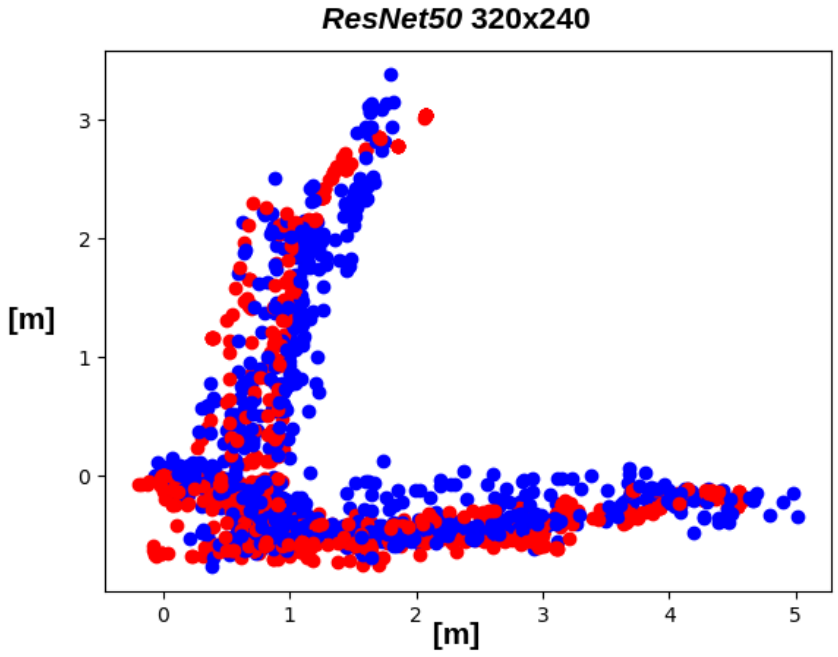


Figura 4.7 – Gráfico de dispersão de pontos correspondente ao modelo *ResNet* com resolução de entrada 320x240. Pontos correspondentes ao *ground truth* em vermelho e pontos de predição em azul.

No retreinamento, também avaliamos o modelo treinado através de gráficos de dispersão.

A Figura 4.7 demonstra o gráfico de dispersão do novo modelo, a fim de compará-lo com o gráfico de dispersão do modelo treinado anteriormente, apresentado na Figura 4.3.

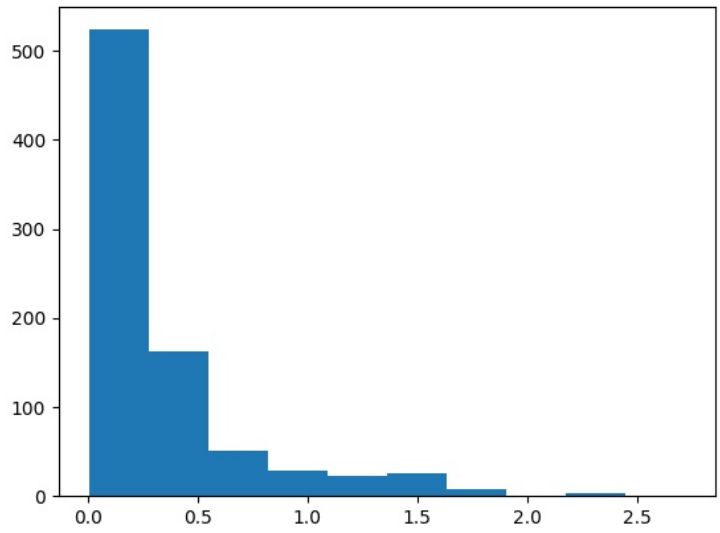


Figura 4.8 – Histograma do modelo *ResNet* retreinado. Eixo y corresponde ao número de amostras e o eixo x corresponde ao erro médio pertencente às amostras.

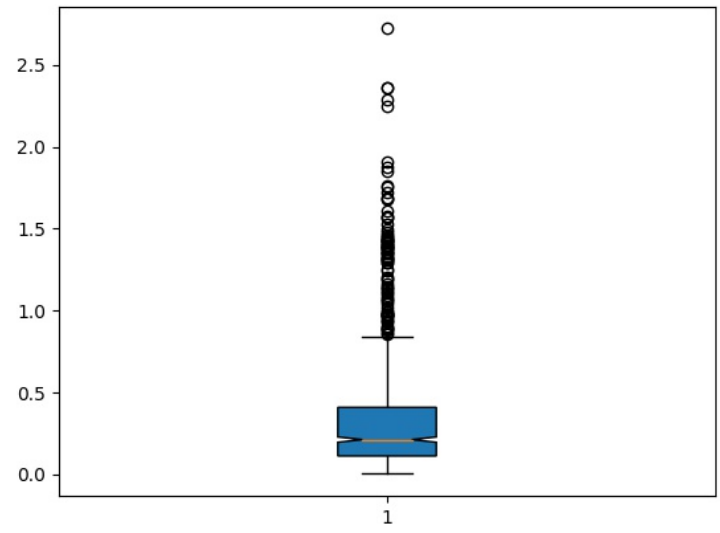


Figura 4.9 – Gráfico *boxplot* do modelo *ResNet* retreinado. Eixo y corresponde ao valor de erro. A caixa azul corresponde ao intervalo da maioria dos valores no conjunto de dados. A linha amarela corresponde à mediana dos erros de predição. Os círculos pretos são os *outliers*.

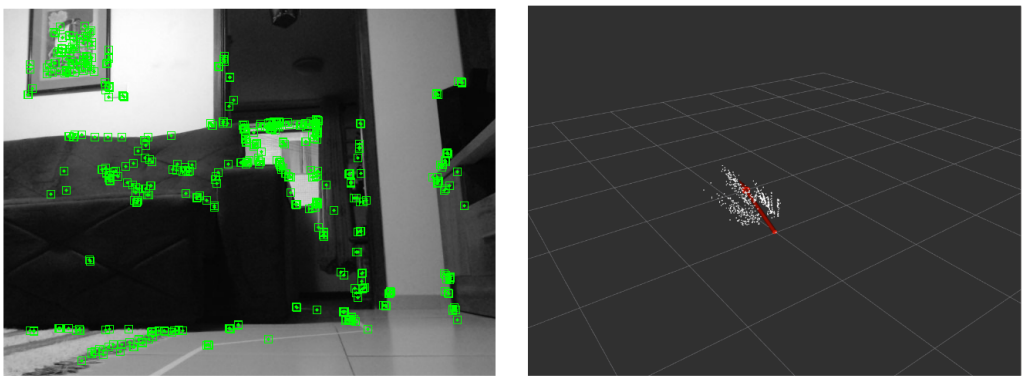
Como observamos nos gráficos de dispersão nas Figuras 4.7 e 4.3, vemos que as predições (em azul) de coordenadas do modelo retreinado se ajusta melhor aos pontos do *ground truth*

(em vermelho). Após a validação visual, também avaliamos o histograma e o *boxplot* do modelo retreinado. As Figuras 4.8 e 4.9 apresentam o histograma e *boxplot* do modelo retreinado.

Observando o histograma do modelo anterior (apresentado na Figura 4.4), observamos que comparado ao do modelo retreinado, temos pelo menos mais de 100 amostras com menor erro. Também observamos a redução de erros individuais (*outliers*) maiores que 1.5 metros. Com relação ao *boxplot*, observamos que a mediana dos erros foi diminuída e também nosso erro médio máximo está abaixo de 1 metro, enquanto no modelo anterior ele estava acima desse valor.

4.2 Comparação com *Orb-Slam2*

Como já mencionado no Capítulo 2, o algoritmo *Orb-Slam*, é um algoritmo de SLAM visual que utiliza câmeras como entrada para seu processamento. Existem duas versões do *Orb-Slam*, uma que utiliza câmeras monoculares (MUR-ARTAL; MONTIEL; TARDÓS, 2015) e outra que utiliza câmeras stereo (MUR-ARTAL; TARDÓS, 2017). Como estamos trabalhando com odometria visual através de câmeras monoculares, vamos comparar a nossa abordagem apenas com a versão monocular do *Orb-Slam*. No entanto, a versão stereo do *Orb-Slam* apresenta resultados melhores com relação à versão monocular.



(a) **(b)**

Figura 4.10 – Em (a) os pontos verdes correspondem aos pontos de referência para estimativa de profundidade e movimento. Em (b) temos os pontos verdes estimados em uma nuvem de pontos 3D, a seta vermelha representa a orientação do robô.

O *Orb-Slam* é um algoritmo de SLAM e não um algoritmo de odometria. No entanto, para a parte de localização do SLAM é necessário estimar a posição do robô dentro do mapa de SLAM. Logo, utilizamos esse dado de posicionamento do *Orb-Slam* para compararmos os resultados. No caso do *Orb-Slam*, as posições são estimadas através da definição de pontos de interesse na imagem, como na Figura 4.10 **(a)**. Quando esses pontos são definidos, a profundidade de cada um deles é estimada, então se cria uma nuvem de pontos como mostrado na Figura 4.10 **(b)**. Esses pontos são utilizados para estimar a distância e movimento entre os frames capturados.

A partir dessa estimativa é calculada uma posição e uma orientação no ambiente. Uma das limitações dessa abordagem é a impossibilidade de se estimar um deslocamento real em metros ou centímetros, pois não existe uma relação conhecida entre os pontos definidos na imagem (pixels) e o mundo real. Logo, os pontos estimados nessa abordagem são todos virtuais, não representando assim deslocamentos no ambiente.

A metodologia do teste consiste em criar uma rota no ambiente real e segui-la com o robô. A primeira rota foi criada com o pacote de odometria *rf2o*, como abordamos na metodologia, esse pacote é utilizado como o *ground truth* de nossa rede neural de odometria. Posteriormente, utilizamos as predições da nossa rede neural de odometria, e depois utilizamos o algoritmo *Orb-Slam*. A comparação das três abordagens pode ser observada na Figura 4.11. Como a rede *ResNet* foi treinada com base no pacote *rf2o*, elas estão na mesma escala. No entanto, as coordenadas geradas pelo *Orb-Slam* são dadas em uma escala virtual. Como câmeras monoculares não possuem relações com o medidas do ambiente real, não é possível estimar o deslocamento com precisão métrica como no caso do pacote *rf2o*, que utiliza lasers. Logo, se fez necessário a normalização dos dados de coordenadas geradas pelo *Orb-Slam* para que as diferentes abordagens pudessem ser comparadas.

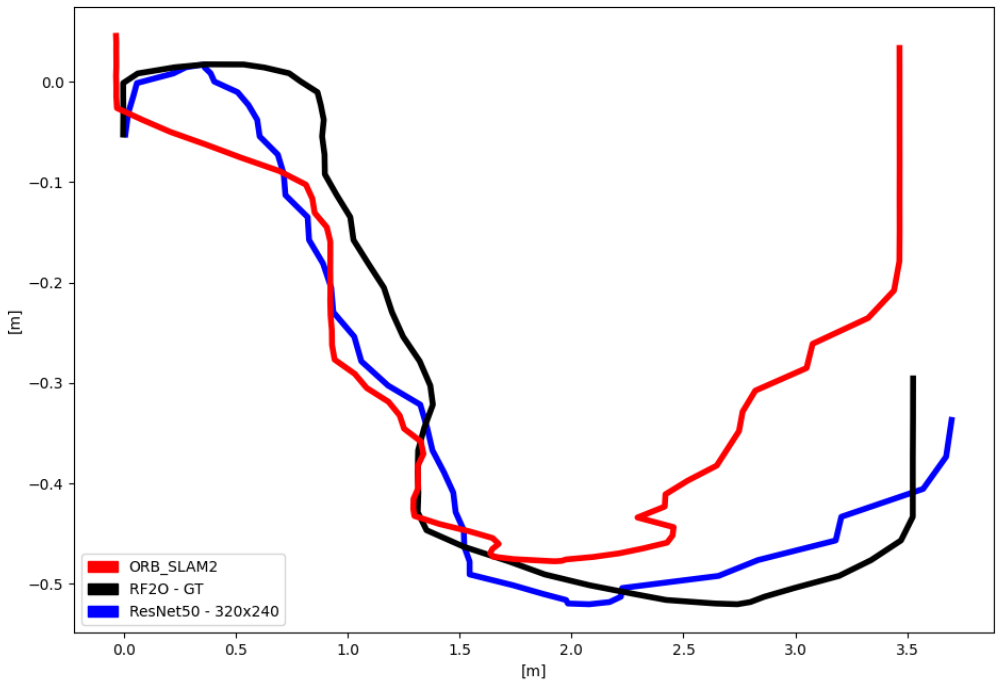


Figura 4.11 – Comparação de precisão das rotas, em preto temos o *ground truth* baseado no pacote *rf2o*. Em azul temos as predições da rede *ResNet*. Em vermelho as estimativas de posição do algoritmo *Orb-Slam* monocular.

Como observamos na Figura 4.11 nossa abordagem obteve um resultado visualmente

melhor quando comparada ao algoritmo *Orb-Slam*. O erro médio obtido pela rede *ResNet* foi de 25.47 centímetros, com relação ao *ground truth* e o erro médio do algoritmo Orb-Slam com relação ao *ground truth* foi de 61.81 centímetros. Além disso, também temos a vantagem de que as predições do modelo *ResNet* são dadas em uma escala métrica, uma vez que são baseadas em medições do laser, através do pacote *rf2o*.

4.3 Avaliação de Desempenho do Modelo Retreinado

Como abordamos um problema de *edge-ai*, avaliamos a performance do modelo *ResNet* em plataformas embarcadas. Essas plataformas são amplamente utilizadas em aplicações em *edge* pois possuem bom poder de processamento com consumo energético razoável. Escolhemos três plataformas embarcadas para os testes, são elas: Raspberry Pi 4, NVIDIA Jetson Nano e Jetson TX2 NX. As plataformas Jetson da NVIDIA são plataformas que possuem vantagens para execução de modelos de aprendizado profundo, pois possuem GPU's com suporte ao CUDA. O CUDA é uma biblioteca e conjunto de ferramentas de desenvolvimento voltado para aplicações em GPU. Essa biblioteca habilita a aceleração de inferência em modelos de aprendizado profundo através das interfaces presentes no *framework TensorFlow*. Com isso, é esperado um melhor desempenho nessas plataformas, em detrimento à Raspberry, que possui apenas módulos do *TensorFlow* que são executados em CPU.

Com relação ao hardware embarcado nas plataformas, todos possuem CPU's na arquitetura ARM. A Raspberry Pi 4, conta com um processador Quad Core Cortex-A72 de 64-bit e 1.5GHz. A plataforma Jetson Nano possui uma CPU Quad-core ARM Cortex-A57 de 64-bit e 1.43GHz e uma GPU com arquitetura Maxwell e 128 CUDA cores. Por fim, a Jetson TX2 NX possui duas CPU's, uma Dual-core NVIDIA Denver 2 de 64-bit e uma Quad Core ARM A57, além das duas CPU's, ela também possui uma GPU com arquitetura Pascal e 256 CUDA cores. Os CUDA cores são os núcleos de processamento gráfico da NVIDIA e quanto mais núcleos, maior o poder de processamento da GPU. Todos os embarcados possuem 4GB de memória RAM disponíveis, que no caso das Jetson são compartilhados entre CPU e GPU.

Avaliamos o consumo de recursos de hardware entre as plataformas e também quantos quadros por segundo (FPS), cada uma das plataformas consegue atingir. A métrica de FPS é útil para avaliação de aplicações em tempo real, onde uma decisão do robô tem um requisito de tempo pré-definido que deve ser respeitado. Calculamos a taxa de FPS de acordo com o tempo em que a plataforma embarcada leva para analisar uma imagem (frame) e retornar uma posição (x,y) dentro do ambiente, através da rede neural. A Tabela 4.3 demonstra o tempo médio de inferência e taxa média de FPS obtida em cada uma das plataformas.

Observamos na Tabela 4.3 que o melhor desempenho é alcançado pela plataforma Jetson TX2, como era esperado pois possui o melhor hardware. No entanto, é notável a melhora nos tempos de inferência quando os mesmos são realizados na GPU, como é o caso das Jetson

Plataforma	Tempo de inferência (s)	FPS
Raspberry Pi 4	4.3	0.23
NVIDIA Jetson Nano	0.226	4.42
NVIDIA Jetson TX2	0.173	5.78

Tabela 4.3 – Tabela de comparação de desempenho em tempo de inferência e FPS das plataformas embarcadas analisadas.

comparadas a Raspberry. A melhora chega 95% a mais de desempenho na TX2 comparada à Raspberry. Todos os testes forma realizados com a versão 2.4.1 do *TensorFlow*, no entanto como mencionado anteriormente, a Raspberry possui suporte apenas a versão de CPU do *framework*, logo a performance é afetada. Os testes realizados na Jetson TX2 NX foram realizados com a CPU Denver desativada devido a um problema descrito pelo fabricante em que os kernels CUDA aumentam a latência quando essa CPU é ativada. Em nossos testes práticos, quando essa CPU era ativada, os tempos de inferência eram negativamente alterados, chegando a valores próximos a 0.303 segundos para inferência. Esse aumento corresponde a um impacto de 42.75% no tempo de inferência.

Também realizamos um comparativo de consumo do hardware dos sistemas embarcados. Essa análise é útil em casos onde será necessário a execução de múltiplas aplicações no mesmo dispositivo embarcado. O script de teste desenvolvido para esse fim é bem simples, ele captura uma imagem pela câmera do robô e passa esse frame para inferência no *TensorFlow*. Os aspectos de consumo de hardware avaliados se baseiam no consumo de memória RAM, CPU e GPU. O consumo de GPU é analisado apenas nos casos das plataformas Jetson, pois somente no caso dessas plataformas a GPU é utilizada.

Na Figura 4.12 observamos o consumo médio de CPU e memória ao executar o processo de inferência do modelo *ResNet*. O consumo de CPU flutuou dos 70 aos 85% de uso, enquanto o consumo de memória ficou abaixo dos 20%. O pico de uso de memória chegou a 540MB ocupados, enquanto o sistema ocioso ocupava cerca de 240MB, tendo um aumento de 125% quando o modelo é executado.

Na Figura 4.13 é observado o uso médio CPU, GPU e memória ao executar o processo de inferência do modelo *ResNet*. O consumo de CPU fica abaixo dos 20%. No entanto o consumo de memória fica perto do limite, em torno de 75%, o que se traduz em 3GB de uso, enquanto o sistema ocioso ocupa 400MB de memória.

Na Figura 4.14 é apresentado o consumo médio de CPU, GPU e memória no processo de inferência do modelo *ResNet*. O consumo de CPU fica abaixo dos 20% e o consumo de memória se estabelece nos 80%. O consumo de memória com o sistema ocioso é de 500MB e quando o processo de inferência é inicializado, temos um uso de memória correspondente a 3.5GB.

O consumo de CPU na Raspberry foi consideravelmente alto, logo no caso da necessidade

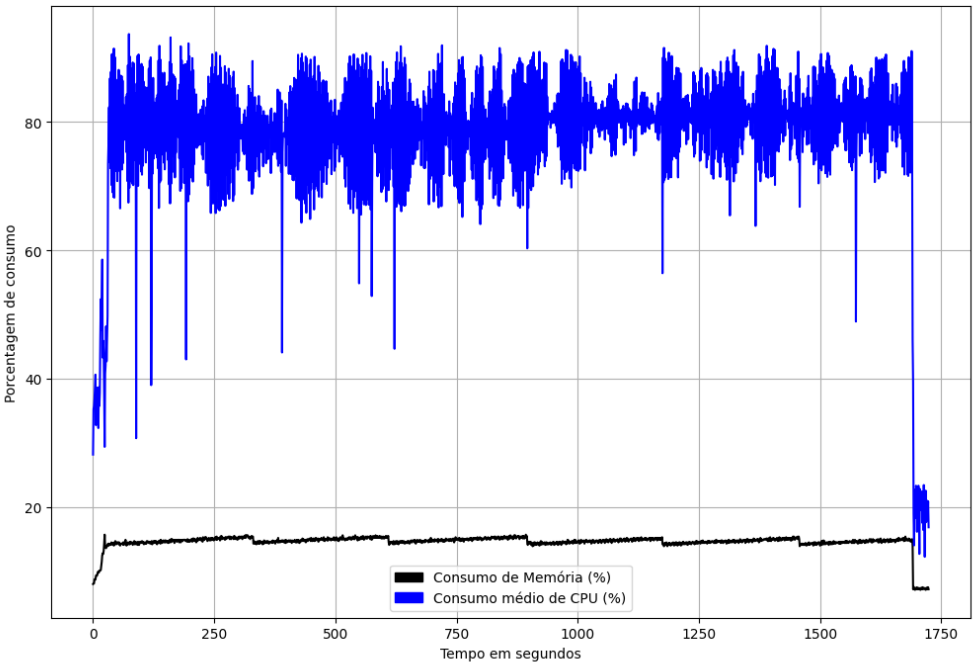


Figura 4.12 – Consumo de hardware na Raspberry Pi 4 ao executar o processo de inferência do modelo *ResNet*. Em azul, o consumo médio de CPU e em preto o consumo de memória.

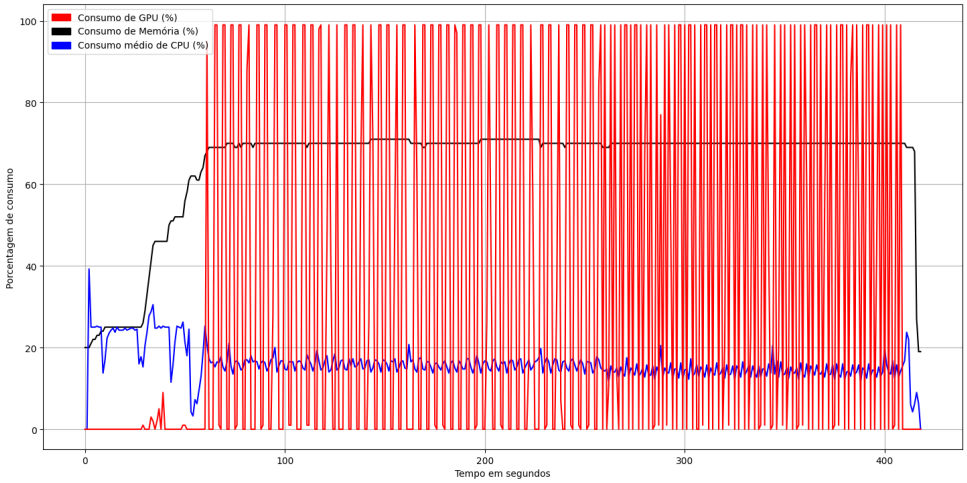


Figura 4.13 – Consumo de hardware na Jetson Nano ao executar o processo de inferência do modelo *ResNet*. Em azul, o consumo médio de CPU, em preto o consumo de memória e em vermelho o consumo de GPU.

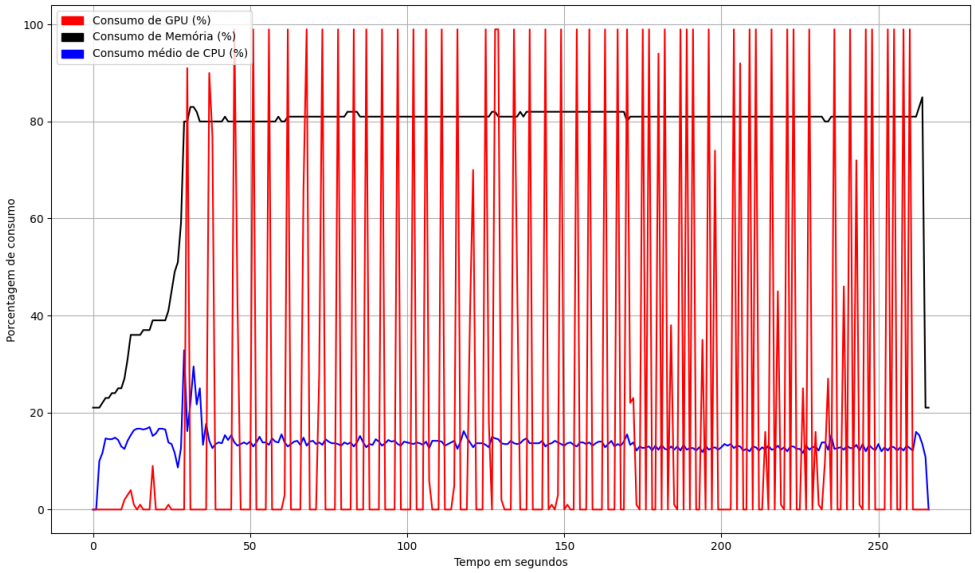


Figura 4.14 – Consumo de hardware na Jetson TX2 NX ao executar o processo de inferência do modelo *ResNet*. Em azul, o consumo médio de CPU, em preto o consumo de memória e em vermelho o consumo de GPU.

de se executar outras aplicações juntas seria necessário analisar o consumo de CPU das outras aplicações. No entanto, o consumo de memória da Raspberry comparado ao consumo das Jetson, podemos observar uma diferença considerável. Essa diferença pode estar relacionada aos módulos extras em que o *TensorFlow* carrega em sua versão de GPU. Como nas plataformas Jetson utilizamos a GPU, a CPU fica mais ociosa comparada à Raspberry. Essa ociosidade na CPU é útil em casos onde é necessário que mais aplicações sejam executadas de forma conjunta. No entanto, o uso de GPU é colocado a 100% toda vez que existe um frame a ser processado. Logo, a GPU está totalmente ocupada com a carga que estamos colocando ao executar o processo de inferência. As grandes diferenças observadas no uso de GPU acontecem pois a GPU só é usada a 100% de uso quando existe um frame a ser processado. Logo, se a aplicação não tem um frame no tempo atual, o uso de GPU é reduzido a 0.

5 Considerações Finais

Neste trabalho, apresentamos uma abordagem de aprendizado profundo para o problema de odometria visual aplicado a um ambiente doméstico mapeado. Coletamos informações básicas relacionando uma imagem a uma posição e orientação no mapa. Também avaliamos três arquiteturas de redes neurais a fim de prever a posição e a orientação do robô no ambiente treinado. Após a validação de uma arquitetura de rede neural convolucional, também testamos A arquitetura *ResNet* em múltiplas plataformas embarcadas com o intuito de avaliar o consumo de hardware e performance dessa arquitetura. Também comparamos a rede neural para cálculos odométricos, com o algoritmo da literatura Orb-Slam. Os resultados obtidos pelos testes indicam uma viabilidade, principalmente nas plataformas Jetson da NVIDIA. No entanto, ainda se faz necessário uma métrica de tempo real para analisar a viabilidade da rede relacionada à taxa de FPS obtida pelos sistemas embarcados analisados.

5.1 Conclusão

Para realizar este trabalho, propusemos uma metodologia utilizando uma plataforma robótica contendo uma Raspberry Pi 4, quatro motores DC com rodas suecas ominidirecionais e uma unidade de medição inercial (IMU). Além disso, a plataforma também usa um Arduino Uno para realizar algumas tarefas computacionais. Também temos um modelo virtual correspondente ao robô real, construído a partir de um URDF utilizando o Robot Operating System (ROS). O robô real, inicialmente realiza a odometria através de um laser planar com o auxílio das medições de uma IMU e relaciona as posições da odometria com a imagens capturadas do ambiente doméstico através de uma câmera embarcada no robô. Os dados gerados por essa abordagem, constroem o conjunto de dados utilizado para treinamento das redes neurais utilizadas.

Para obtermos um modelo de rede neural aplicado a odometria visual, coletamos 6203 imagens. Em seguida, propomos o uso de três arquiteturas de redes neurais convolucionais diferentes (CNNs). No início, tentamos usar um modelo CNN personalizado, com cinco camadas convolucionais alternadas com camadas de max-pooling 2x2. Posteriormente, realizamos experimentos com um modelo utilizando o *backbone VGG-16*, alterando apenas as camadas de saída da rede para obtermos os resultados esperados. Por fim, também realizamos experimentos em modelo com o *ResNet50*. O último modelo obteve os melhores resultados e foi analisado com relação ao seu histograma de erros médios a fim de se analisara convergência.

No primeiro processo de avaliação dos modelos, mostrou que o modelo *ResNet50* foi capaz de aprender melhor as características do ambiente doméstico utilizado nos experimentos. Este modelo apresentou a menor taxa de erro entre os três modelos avaliados. O erro médio de 45.87 centímetros ainda era alto, considerando o tamanho total do ambiente de 4.5758 metros. Para

melhorar isso, retreinamos o modelo *ResNet* com mais 1823 imagens, além de mudarmos alguns parâmetros relacionados aos dados de entrada do dataset. Com o retreinamento, diminuímos o erro médio do modelo de 45.87 centímetros para 35.19 centímetros, o que ocasionou em 7.68% da maior distância no ambiente de testes.

Posteriormente, comparamos a solução de rede neural de odometria com o algoritmo da literatura Orb-Slam. Visualmente obtivemos um resultado melhor, no entanto, também avaliamos a distância de cada coordenada de previsão com relação ao *ground truth* e também avaliamos a distância das coordenadas do Orb-Slam com relação ao *ground truth*. Nesta avaliação, obtivemos o resultado de 25.47 centímetros de erro médio para as previsões da rede *ResNet* e uma média de erro de 61.81 centímetros para o Orb-Slam.

Por último, avaliamos o consumo de hardware relacionado à taxa de FPS obtida em diferentes plataformas embarcadas. A comparação foi realizada nos hardwares embarcados: Raspberry Pi 4, Jetson Nano e Jetson TX2. Os testes foram feitos em sistemas operacionais Ubuntu para hardwares ARM, com versão do *TensorFlow* 2.4.1. Os testes demonstraram que os melhores resultados com relação a métrica de FPS são alcançados nas plataformas Jetson, algo que já era esperado devido a presença do *TensorFlow* para GPU's. A Raspberry também conta com uma GPU, no entanto, ela não é suportada pelo *TensorFlow*. Logo, o processo de inferência na Raspberry foi feito diretamente na CPU. Os resultados mostram uma performance de 5.78 FPS na Jetson TX2, 4.42 FPS na Jetson Nano e 0.23 FPS na Raspberry. Para uma melhor avaliação desses resultados, é necessário a definição de um requisito mínimo de tempo para o processamento dessas imagens. Esse requisito de tempo, pode variar de aplicação para aplicação dependendo de quão crítica é a tarefa de odometria. Também avaliamos o consumo de hardware em todas as três plataformas embarcadas. No embarcado Raspberry, o consumo de CPU flutuou entre os 70 aos 85% de uso, enquanto o uso de memória ficou em 540MB. Nas plataformas Jetson, a GPU é estressada ao limite, chegando ao total de seu uso quando se tem uma imagem a ser processada. Apesar do maior desempenho, relacionado à Raspberry, o consumo de memória nas plataformas Jetson é consideravelmente maior, passando dos 3GB de utilização. Uma explicação desse fato, pode ser o maior número de bibliotecas importadas pelo *framework TensorFlow* em sua versão de GPU. O aumento do uso de memória das Jetson com relação a Raspberry chega a 548.15%, com o *tradeoff* de um aumento considerável de 2413% na performance com relação ao FPS.

5.2 Trabalhos Futuros

Em futuras melhorias deste trabalho, se faz necessário a avaliação de uma métrica para validar as informações de orientação da rede. Essa informação de orientação é gerada como saída da rede, no entanto não avaliamos a precisão dessa predição por não chegarmos em uma métrica de avaliação. Outra melhoria futura é um método de validação das medições do LIDAR

por meio de um algoritmo de visão computacional. Esse algoritmo de visão relacionaria uma posição conhecida no ambiente à posição de saída do LIDAR, para que assim as posições em mais coordenadas pudessem ser validadas. E por fim, coletar mais dados para a alimentação do treinamento da rede neural. Observamos na literatura, várias abordagens de treinamentos de redes neurais que utilizam maiores números de imagens para um treinamento aceitável. Devido a restrições de tempo, a coleta de um maior número de imagens não foi possível e como mostrado no retreinamento pode beneficiar a aplicação. Além dessas adições, também é interessante medir o consumo energético de cada plataforma embarcada, visto que sistemas de *edge* tem uma tendência de serem alimentados por baterias. Logo, em uma futura versão deste trabalho é interessante a adição de uma comparação de consumo energético da execução do modelo nas diferentes plataformas embarcadas utilizadas.

5.3 Publicações Realizadas

O trabalho seguinte foi concebido através da metodologia proposta pelo presente trabalho. O trabalho foi aceito e apresentado na conferência nacional:

1. Frederico Luiz Martins de Sousa, Natália F. de C. Meira, Mateus Coelho Silva, Ricardo Augusto Rabelo Oliveira. Deep-Learning-Based Odometry Models for Mobile Robotics. Florianópolis, Brasil: XI Brazilian Symposium on Computing Systems Engineering - SBESC 2021, 2021.

Referências

ALATISE, M. B.; HANCKE, G. P. Pose estimation of a mobile robot based on fusion of imu data and vision data using an extended kalman filter. *Sensors*, Multidisciplinary Digital Publishing Institute, v. 17, n. 10, p. 2164, 2017.

AZPURUA, H.; REZENDE, A.; POTJE, G.; JÚNIOR, G. P. d. C.; FERNANDES, R.; MIRANDA, V.; DOMINGUES, J.; ROCHA, F.; SOUSA, F. L. M. de; BARROS, L. G. D. de et al. Towards semi-autonomous robotic inspection and mapping in confined spaces with the espeleorobô. *Journal of Intelligent & Robotic Systems*, Springer, v. 101, n. 4, p. 1–27, 2021.

BENDER, D.; KOCH, W.; CREMERS, D. Map-based drone homing using shortcuts. In: IEEE. *2017 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI)*. [S.l.], 2017. p. 505–511.

CHENG, Y.; MAIMONE, M.; MATTHIES, L. Visual odometry on the mars exploration rovers. In: IEEE. *2005 IEEE International Conference on Systems, Man and Cybernetics*. [S.l.], 2005. v. 1, p. 903–910.

CHOI, M.-K.; JUNG, H. Development of fast refinement detectors on ai edge platforms. In: SPRINGER. *International Conference on Pattern Recognition*. [S.l.], 2021. p. 592–606.

CID, A.; NAZÁRIO, M.; SATHLER, M.; MARTINS, F.; DOMINGUES, J.; DELUNARDO, M.; ALVES, P.; TEOTÔNIO, R.; BARROS, L. G.; REZENDE, A. et al. A simulated environment for the development and validation of an inspection robot for confined spaces. In: IEEE. *2020 Latin American Robotics Symposium (LARS), 2020 Brazilian Symposium on Robotics (SBR) and 2020 Workshop on Robotics in Education (WRE)*. [S.l.], 2020. p. 1–6.

FUKUSHIMA, K. Cognitron: A self-organizing multilayered neural network. *Biological cybernetics*, Springer, v. 20, n. 3, p. 121–136, 1975.

GEIGER, A.; ZIEGLER, J.; STILLER, C. Stereoscan: Dense 3d reconstruction in real-time. In: IEEE. *2011 IEEE intelligent vehicles symposium (IV)*. [S.l.], 2011. p. 963–968.

GIUBILATO, R.; CHIODINI, S.; PERTILE, M.; DEBEI, S. An evaluation of ros-compatible stereo visual slam methods on a nvidia jetson tx2. *Measurement*, Elsevier, v. 140, p. 161–170, 2019.

GREHL, S.; DONNER, M.; FERBER, M.; DIETZE, A.; MISCHO, H.; JUNG, B. Mining-rox–mobile robots in underground mining. In: *Proceedings of the Third International Future Mining Conference, Sydney, Australia*. [S.l.: s.n.], 2015. p. 4–6.

GRISETTI, G.; STACHNISS, C.; BURGARD, W. Improved techniques for grid mapping with rao-blackwellized particle filters. *IEEE transactions on Robotics*, IEEE, v. 23, n. 1, p. 34–46, 2007.

HE, K.; ZHANG, X.; REN, S.; SUN, J. Deep residual learning for image recognition. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. [S.l.: s.n.], 2016. p. 770–778.

JAIMEZ, M.; MONROY, J. G.; GONZÁLEZ-JIMÉNEZ, J. Planar odometry from a radial laser scanner. a range flow-based approach. In: *IEEE International Conference on Robotics and Automation (ICRA)*. [s.n.], 2016. p. 4479–4485. Disponível em: <http://mapir.isa.uma.es/mapirwebsite/index.php/mapir-downloads/papers/217>.

JAULIN, L. *Mobile robotics*. [S.l.]: John Wiley & Sons, 2019.

KHAN, W. Z.; AHMED, E.; HAKAK, S.; YAQOOB, I.; AHMED, A. Edge computing: A survey. *Future Generation Computer Systems*, Elsevier, v. 97, p. 219–235, 2019.

KLIPPEL, E.; OLIVEIRA, R.; MASLOV, D.; BIANCHI, A.; SILVA, S. E.; GARROCHO, C. Towards to an embedded edge ai implementation for longitudinal rip detection in conveyor belt. In: SBC. *Anais Estendidos do X Simpósio Brasileiro de Engenharia de Sistemas Computacionais*. [S.l.], 2020. p. 97–102.

KOHLBRECHER, S.; MEYER, J.; STRYK, O. von; KLINGAUF, U. A flexible and scalable slam system with full 3d motion estimation. In: IEEE. *Proc. IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*. [S.l.], 2011.

KRIZHEVSKY, A.; SUTSKEVER, I.; HINTON, G. E. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, v. 25, p. 1097–1105, 2012.

LEUTENEGGER, S.; LYNEN, S.; BOSSE, M.; SIEGWART, R.; FURGALE, P. Keyframe-based visual–inertial odometry using nonlinear optimization. *The International Journal of Robotics Research*, SAGE Publications Sage UK: London, England, v. 34, n. 3, p. 314–334, 2015.

LI, E.; ZENG, L.; ZHOU, Z.; CHEN, X. Edge ai: On-demand accelerating deep neural network inference via edge computing. *IEEE Transactions on Wireless Communications*, IEEE, v. 19, n. 1, p. 447–457, 2019.

LI, R.; WANG, S.; LONG, Z.; GU, D. Undeepvo: Monocular visual odometry through unsupervised deep learning. In: IEEE. *2018 IEEE international conference on robotics and automation (ICRA)*. [S.l.], 2018. p. 7286–7291.

LIN, S.; ZHOU, Z.; ZHANG, Z.; CHEN, X.; ZHANG, J. Edge intelligence in the making: Optimization, deep learning, and applications. *Synthesis Lectures on Learning, Networks, and Algorithms*, Morgan & Claypool Publishers, v. 1, n. 2, p. 1–233, 2020.

LIN, X.; LI, J.; WU, J.; LIANG, H.; YANG, W. Making knowledge tradable in edge-ai enabled iot: A consortium blockchain-based efficient and incentive approach. *IEEE Transactions on Industrial Informatics*, IEEE, v. 15, n. 12, p. 6367–6378, 2019.

LIU, M.; NIU, J.; WANG, X. An autopilot system based on ros distributed architecture and deep learning. In: IEEE. *2017 IEEE 15th International Conference on Industrial Informatics (INDIN)*. [S.l.], 2017. p. 1229–1234.

MAZZIA, V.; KHALIQ, A.; SALVETTI, F.; CHIABERGE, M. Real-time apple detection system using embedded systems with hardware accelerators: An edge ai application. *IEEE Access*, IEEE, v. 8, p. 9102–9114, 2020.

MCCULLOCH, W. S.; PITTS, W. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, Springer, v. 5, n. 4, p. 115–133, 1943.

MOHAMED, S. A.; HAGHBAYAN, M.-H.; WESTERLUND, T.; HEIKKONEN, J.; TENHUNEN, H.; PLOSILA, J. A survey on odometry for autonomous navigation systems. *IEEE Access*, IEEE, v. 7, p. 97466–97486, 2019.

MUNERA, E.; POZA-LUJAN, J.-L.; POSADAS-YAGUE, J.-L.; SIMO, J.; NOGUERA, J. F. B. Distributed real-time control architecture for ros-based modular robots. *IFAC-PapersOnLine*, Elsevier, v. 50, n. 1, p. 11233–11238, 2017.

MUR-ARTAL, R.; MONTIEL, J. M. M.; TARDÓS, J. D. ORB-SLAM: a versatile and accurate monocular SLAM system. *IEEE Transactions on Robotics*, v. 31, n. 5, p. 1147–1163, 2015.

MUR-ARTAL, R.; TARDÓS, J. D. ORB-SLAM2: an open-source SLAM system for monocular, stereo and RGB-D cameras. *IEEE Transactions on Robotics*, v. 33, n. 5, p. 1255–1262, 2017.

NOSRATABADI, S.; MOSAVI, A.; DUAN, P.; GHAMISI, P.; FILIP, F.; BAND, S. S.; REUTER, U.; GAMA, J.; GANDOMI, A. H. Data science in economics: comprehensive review of advanced machine learning and deep learning methods. *Mathematics*, Multidisciplinary Digital Publishing Institute, v. 8, n. 10, p. 1799, 2020.

QUIGLEY, M.; CONLEY, K.; GERKEY, B.; FAUST, J.; FOOTE, T.; LEIBS, J.; WHEELER, R.; NG, A. Y. et al. Ros: an open-source robot operating system. In: KOBE, JAPAN. *ICRA workshop on open source software*. [S.l.], 2009. v. 3, n. 3.2, p. 5.

RUBLEE, E.; RABAUD, V.; KONOLIGE, K.; BRADSKI, G. Orb: An efficient alternative to sift or surf. In: IEEE. *2011 International conference on computer vision*. [S.l.], 2011. p. 2564–2571.

SAGAN, C. *Pale Blue Dot*. 1994.

SANTOS, J. M.; PORTUGAL, D.; ROCHA, R. P. An evaluation of 2d slam techniques available in robot operating system. In: IEEE. *2013 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*. [S.l.], 2013. p. 1–6.

SHAN, T.; ENGLOT, B. Lego-loam: Lightweight and ground-optimized lidar odometry and mapping on variable terrain. In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. [S.l.: s.n.], 2018. p. 4758–4765.

SHARMA, R.; BIOOKAGHAZADEH, S.; LI, B.; ZHAO, M. Are existing knowledge transfer techniques effective for deep learning with edge devices? In: IEEE. *2018 IEEE International Conference on Edge Computing (EDGE)*. [S.l.], 2018. p. 42–49.

SHI, W.; DUSTDAR, S. The promise of edge computing. *Computer*, IEEE, v. 49, n. 5, p. 78–81, 2016.

SHI, Y.; YANG, K.; JIANG, T.; ZHANG, J.; LETAIEF, K. B. Communication-efficient edge ai: Algorithms and systems. *IEEE Communications Surveys & Tutorials*, IEEE, v. 22, n. 4, p. 2167–2191, 2020.

SILVA, M. C.; SOUSA, F. L. M. de; BARBOSA, D. L. M.; OLIVEIRA, R. A. R. Constraints and challenges in designing applications for industry 4.0: A functional approach. In: *ICEIS (1)*. [S.l.: s.n.], 2020. p. 767–774.

SIMONYAN, K.; ZISSERMAN, A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

SOUSA, F. L. M. de; SILVA, M. J. da; SANTOS, R. C. C. de M.; SILVA, M. C.; OLIVEIRA, R. A. R. Deep-learning-based embedded adas system. In: IEEE. *2021 XI Brazilian Symposium on Computing Systems Engineering (SBESC)*. [S.l.], 2021. p. 1–8.

TAKETOMI, T.; UCHIYAMA, H.; IKEDA, S. Visual slam algorithms: a survey from 2010 to 2016. *IPSJ Transactions on Computer Vision and Applications*, SpringerOpen, v. 9, n. 1, p. 1–11, 2017.

VOULODIMOS, A.; DOULAMIS, N.; DOULAMIS, A.; PROTOPAPADAKIS, E. Deep learning for computer vision: A brief review. *Computational intelligence and neuroscience*, Hindawi, v. 2018, 2018.

WANG, X.; HAN, Y.; LEUNG, V. C.; NIYATO, D.; YAN, X.; CHEN, X. Convergence of edge computing and deep learning: A comprehensive survey. *IEEE Communications Surveys & Tutorials*, IEEE, v. 22, n. 2, p. 869–904, 2020.

YAN, M.; WANG, J.; LI, J.; ZHANG, C. Loose coupling visual-lidar odometry by combining viso2 and loam. In: IEEE. *2017 36th Chinese Control Conference (CCC)*. [S.l.], 2017. p. 6841–6846.

ZHAN, H.; WEERASEKERA, C. S.; BIAN, J.-W.; REID, I. Visual odometry revisited: What should be learnt? In: IEEE. *2020 IEEE International Conference on Robotics and Automation (ICRA)*. [S.l.], 2020. p. 4203–4210.

ZHANG, J.; SINGH, S. Loam: Lidar odometry and mapping in real-time. In: *Robotics: Science and Systems*. [S.l.: s.n.], 2014. v. 2, n. 9.