UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE GRADUAÇÃO EM ENGENHARIA DE CONTROLE E AUTOMAÇÃO

Linda Dotto de Moraes

COMPARAÇÃO DE ALGORITMOS DE APRENDIZAGEM POR REFORÇO PROFUNDO NA NAVEGAÇÃO DO ROBÔ MÓVEL E DESVIO DE TRAJETÓRIA

Santa Maria, RS
2022

Linda Dotto de Moraes

COMPARAÇÃO DE ALGORITMOS DE APRENDIZAGEM POR REFORÇO PROFUNDO NA NAVEGAÇÃO DO ROBÔ MÓVEL E DESVIO DE TRAJETÓRIA

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Controle e Automação, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Engenheira em Engenharia de Controle e Automação**.

ORIENTADOR: Prof. Daniel Fernando Tello Gamarra

Santa Maria, RS
2022

Linda Dotto de Moraes

COMPARAÇÃO DE ALGORITMOS DE APRENDIZAGEM POR REFORÇO PROFUNDO NA NAVEGAÇÃO DO ROBÔ MÓVEL E DESVIO DE TRAJETÓRIA

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Controle e Automação, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Engenheira em Engenharia de Controle e Automação**.

Aprovado em 23 de setembro de 2022:

Daniel Fernando Tello Gamarra, Prof. Dr. (UFSM)
(Presidente/Orientador)

Anselmo Rafael Cukla, Prof. Dr. (UFSM)

Marcelo Serrano Zanetti, Prof. Dr. (UFSM)

Santa Maria, RS
2022

AGRADECIMENTOS

Meus agradecimentos vão primeiramente a meus pais, que deram suporte à busca pelo diploma, sempre estimulando que tomasse minhas próprias escolhas. Agradeço a toda base de criatividade, raciocínio lógico e consciência social que sempre despertaram em mim.

Também agradeço imensamente à Júlia que entrou junto comigo nesta jornada de Engenharia e me acompanhou em quase todas as empreitadas acadêmicas possíveis. Agradeço aos colegas e amigos Érico, Alison, Gabriel e Geise, por participarem desse período de graduação de forma única, proporcionando boas risadas mas também compartilhando muito conhecimento e horas de estudo na biblioteca. Ao meu parceiro Gabriel, que foi suporte e incentivo a todos os momentos, sempre impulsionando minhas capacidades e acolhendo meus medos. Aos meus amigos e familiares mais próximos, incabíveis de citar aqui mas que me acompanharam em toda a minha trajetória acadêmica de forma indireta, meu eterno agradecimento.

Não posso deixar de agradecer ao Professor Gamarra, que me deu acolheu como orientada nesse projeto e teve tranquilidade em me orientar em todas as reuniões a distância que tivemos, muitas vezes com o emocional abalado pela pandemia que enfrentamos durante o período que trabalhamos juntos.

Também devo agradecer ao grupo GARRA por disponibilizar material e pessoal para me apoiar nessa pesquisa, em especial ao Victor, Alisson, Jair e Emerson que estiveram comigo nas fases crucias desse trabalho.

Deixo meu muito obrigada a Universidade Federal de Santa Maria, essa instituição de educação pública, gratuita e de qualidade, em que tive a oportunidade de obter minha formação de nível médio, técnico e superior. Quando falo sobre a UFSM, estendo meus agradecimentos a todos os professores e técnicos administrativos que a integram e que a fazem ser o que é, contribuindo diariamente para a minha formação e a de tantos outros alunos que por aqui passam.

Que sei eu do que serei, eu que nao sei o que sou? Ser o que penso? Mas penso tanta coisa!

(Fernando Pessoa)

RESUMO

COMPARAÇÃO DE ALGORITMOS DE APRENDIZAGEM POR REFORÇO PROFUNDO NA NAVEGAÇÃO DO ROBÔ MÓVEL E DESVIO DE TRAJETÓRIA

AUTORA: Linda Dotto de Moraes
ORIENTADOR: Daniel Fernando Tello Gamarra

Este trabalho apresenta um estudo de duas abordagens de Deep Reinforcement Learning (Deep-RL) para melhorar o problema de navegação sem mapa para um robô móvel terrestre. A metodologia foca na comparação de uma técnica Deep-RL baseada no algoritmo Deep Q-Network (DQN) com uma segunda baseada no algoritmo Double Deep Q-Network (DDQN). Foram utilizadas 24 leituras do laser, a posição relativa e o ângulo do agente em relação ao alvo como informações para o agente, que fornecem as ações como velocidades para o robô. Ao usar uma estrutura de aprendizado de sensoriamento de baixa dimensão como a proposta, mostra-se que é possível treinar um agente com sucesso para realizar tarefas relacionadas à navegação e evitar obstáculos sem o uso de informações de sensoriamento complexas, como em abordagens baseadas em imagens. A metodologia proposta foi utilizada com sucesso em três ambientes distintos reais e simulados. Assim, foi mostrado que as estruturas Double Deep melhoram ainda mais o problema para a navegação de robôs móveis quando comparados aos de estruturas Q simples.

Palavras-chave: Robótica. Aprendizagem Profunda. Aprendizagem por Reforço Profundo

ABSTRACT

ROBOT NAVIGATION AND OBSTACLE AVOIDANCE USING DEEP REINFORCEMENT LEARNING

AUTHOR: Linda Dotto de Moraes
ADVISOR: Daniel Fernando Tello Gamarra

This work presents two Deep Reinforcement Learning (Deep-RL) approaches to enhance the problem of mapless navigation for a terrestrial mobile robot. The methodology focus on comparing a Deep-RL technique based on the Deep Q-Network (DQN) algorithm with a second one based on the Double Deep Q-Network (DDQN) algorithm. As sensor, 24 laser range findings samples were used and the relative position and angle of the agent to the target were used as information for the agent, which provide the actions as velocities for the robot. By using a low-dimensional sensing structure of learning such as the one proposed, it is shown that it is possible to successfully train an agent to perform navigation-related tasks and obstacle avoidance without the use of complex sensing information, like in image-based approaches. The proposed methodology was successfully used in three distinct real and simulated environments. Overall, it was shown that Double Deep structures further enhance the problem for the navigation of mobile robots when compared to the ones with simple Q structures.

Keywords: Robotics.Deep Learning. Deep Reinforcement Learning.

LISTA DE FIGURAS

Figura 1 – Exemplo de Redes Neurais Convolucionais num problema de classificação 17
Figura 2 – Função de ativação ReLU 17
Figura 3 – Estrutura do Processo de Decisão de Markov 19
Figura 4 – Estrutura de nós do ROS 27
Figura 5 – Estrutura de nós ampliada do ROS com Teleoperação 27
Figura 6 – Simulação de robô manipulador no Gazebo 28
Figura 7 – Ambiente de simulação 1 29
Figura 8 – Ambiente de simulação 2 29
Figura 9 – Ambiente de simulação 3 30
Figura 10 – Ambiente real 1 31
Figura 11 – Ambiente real 2 31
Figura 12 – Ambiente real 3 32
Figura 13 – Jetson Nano B01 com periféricos 33
Figura 14 – Rede Q utilizada 34
Figura 15 – Turtlebot3 36
Figura 16 – Fluxograma das etapas de treinamento e teste simulado e real 37
Figura 17 – Simulação do Turtlebot 3 no gazebo 39
Figura 18 – Média Móvel das recompensas obtidas pelo Turtlebot3 em cada episódio de iteração no Ambiente 1. 39
Figura 19 – Média Móvel das pontuações obtidas pelo Turtlebot3 em cada episódio no Ambiente 2. 40
Figura 20 – Média Móvel das pontuações obtidas pelo Turtlebot3 em cada episódio no Ambiente 3. 40
Figura 21 – Trajetórias realizadas pelo Turtlebot3 no teste em simulação no Ambiente 1 com DQN. 41
Figura 22 – Trajetórias realizadas pelo Turtlebot3 no teste em simulação no Ambiente 1 com DDQN. 42
Figura 23 – Trajetórias realizadas pelo Turtlebot3 no teste em simulação no Ambiente 2 com DQN. 42
Figura 24 – Trajetórias realizadas pelo Turtlebot3 no teste em simulação no Ambiente 2 com DDQN. 43
Figura 25 – Trajetórias realizadas pelo Turtlebot3 no teste em simulação no Ambiente 3 com DQN. 43
Figura 26 – Trajetórias realizadas pelo Turtlebot3 no teste em simulação no Ambiente 3 com DDQN. 44
Figura 27 – Trajetórias realizadas pelo Turtlebot3 no teste no Ambiente 1 real com DQN. 45
Figura 28 – Trajetórias realizadas pelo Turtlebot3 no teste no Ambiente 1 real com DDQN. 45
Figura 29 – Trajetórias realizadas pelo Turtlebot3 no teste no Ambiente 2 real com DQN. 46
Figura 30 – Trajetórias realizadas pelo Turtlebot3 no teste no Ambiente 2 real com DDQN. 46
Figura 31 – Trajetórias realizadas pelo Turtlebot3 no teste no Ambiente 3 real com DQN. 47

Figura 32 – Trajetórias realizadas pelo Turtlebot3 no teste no Ambiente 3 real com DDQN. 47
Figura 33 – Navegação de cada algoritmo em cada ambiente simulado em 100 tentativas. 48
Figura 34 – Navegação de cada algoritmo em cada ambiente real em 4 tentativas. . 48

LISTA DE TABELAS

Tabela 1 – Ação e velocidade angular 34
Tabela 2 – Comparação dos resultados dos testes reais e simulados..... 48
Tabela 3 – Comparação dos resultados dos testes reais e simulados..... 49
Tabela 4 – Comparação dos resultados dos testes reais e simulados..... 49

SUMÁRIO

1	**INTRODUÇÃO**	**10**
1.1	JUSTIFICATIVA	11
1.2	OBJETIVO GERAL E ESPECÍFICOS	12
1.3	DIVISÃO DO TRABALHO	12
2	**REVISÃO DE LITERATURA**	**14**
2.1	TRABALHOS RELACIONADOS	14
2.2	APRENDIZAGEM PROFUNDA	16
2.3	APRENDIZAGEM POR REFORÇO	18
2.4	APRENDIZAGEM POR REFORÇO PROFUNDO	19
2.5	ALGORITMO DE Q-LEARNING	20
2.6	DEEP Q-LEARNING	22
2.7	ALGORITMO DE DOUBLE DEEP Q-LEARNING	23
3	**MATERIAIS E MÉTODOS**	**25**
3.1	DOCKER	25
3.2	PYTORCH	25
3.3	ROBOT OPERATIONAL SYSTEM	26
3.3.1	**Gazebo**	28
3.3.2	**Ambientes de simulação**	28
3.4	AMBIENTES REAIS	30
3.4.1	**Visão Computacional e OpenCV**	32
3.4.2	**Estrutura da Rede Q**	33
3.4.3	**Função de recompensa**	35
3.5	TURTLEBOT	35
3.6	METODOLOGIA DE AVALIAÇÃO DE DESEMPENHO	36
4	**APRESENTAÇÃO E DISCUSSÃO DOS RESULTADOS**	**37**
4.1	SIMULAÇÃO E TREINAMENTO	38
4.2	TESTE EM SIMULAÇÃO E REAL	41
4.3	TESTES REAIS	44
5	**CONCLUSÃO**	**50**
	REFERÊNCIAS BIBLIOGRÁFICAS	**51**
	ANEXO A – CÓDIGO DQN	**53**
	ANEXO B – CÓDIGO DDQN	**61**

1 INTRODUÇÃO

Com o avanço das tecnologias, a inteligência artificial hoje é utilizada nas mais diversas aplicações. A aprendizagem de máquina está presente até mesmo em funções que parecem básicas, como classificação de resultados de pesquisa na internet, reconhecimento de voz dos smartfones e recomendações de vídeos no *Youtube*. Concomitantemente, as aplicações de robótica estão cada vez mais avançadas, seja nas fábricas com alta tecnologia, em que realizam tarefas com precisão, ou dentro dos domicílios, realizando tarefas domésticas simples. Misturando conceitos básicos de robótica com aprendizagem por reforço e aprendizagem profunda, é possível aperfeiçoar o desempenho de robôs em tarefas que antes pareciam tema de filmes de ficção científica.

Dentro da robótica, a utilização de robôs móveis também está cada vez mais abrangente. Em muitos casos, facilitam a execução de tarefas realizadas por humanos, desde tarefas simples como aspirar a poeira da casa, até tarefas mais complexas, como exploração de minas e operações de combate. Esses robôs móveis e autônomos exigem estratégias de navegação e desvio de trajetória.

Para o planejamento dessas estratégias, se utiliza muito a aprendizagem por reforço, quando se trata de tarefas mais simples. O problema dessa técnica é que o robô tem que ser exposto em treinamento a todos os ambientes que pode vir a encarar no mundo real para que aprenda como agir neles. E, mesmo que hoje se tenha mais recursos de computação disponíveis - como CPU, espaço de memória, espaço em disco -, treinar o robô em todos ambientes pode se tornar muito custoso em processamento, recursos e tempo. Nesses casos, a união dos métodos de aprendizagem por reforço junto a técnicas de aprendizagem profunda se torna uma estratégia interessante na solução dos problemas de navegação de robôs móveis.

A aprendizagem profunda é uma área em expansão, sendo muito utilizada em visão computacional, em aplicações como identificação e reconhecimento de objetos em imagens. Um exemplo de aplicação são os terminais de reconhecimento facial. Na pandemia de Covid-19, esses terminais são utilizados inclusive com artifícios de reconhecimento de uso de máscaras de proteção na entrada de ambientes como shoppings.

Inspiradas em neurônios biológicos e suas conexões, as redes neurais artificiais que baseiam a aprendizagem profunda são poderosas, versáteis e escaláveis. Para viabilizar o desenvolvimento e treinamento de redes neurais existem bibliotecas, como o Keras do TensorFlow e o PyTorch. Estas são poderosas bibliotecas de software para cálculo numérico de código aberto especialmente adequada e ajustada para o aprendizado de máquina em larga escala. Para otimizar e agilizar o processamento dos dados, essas bibliotecas permitem até a aplicação de técnicas de computação paralela em várias GPUs e servidores.

Dentro da área de robótica e aprendizagem por reforço profundo, o algoritmo baseado em valor de $Q - Learning$ demonstrou potencial para a navegação de robôs móveis. Além disso, utilizando o meta sistema operacional ROS - *Robot Operating System* -, pode-se dispor de ferramentas que dão suporte a exploração de novas técnicas na área de robótica. Assim, é possível aplicar esses algoritmos e avaliar seu desempenho na navegação de robôs.

1.1 JUSTIFICATIVA

A robótica móvel é uma área em expansão, e tem-se cada vez mais robôs autônomos designados para realização de diferentes tarefas. Dessa forma são necessárias técnicas cada vez mais especializadas que possibilitem a interação dos robôs móveis no mundo real, como a aprimoração dos métodos de desvio de trajetória. Uma possibilidade de solução para problemas de navegação de robôs móveis é através de estratégias de controle clássico, como controle proporcional integral derivativo.

Mas as técnicas de controle clássico não são suficientes quando o robô encara alguma situação não prevista, como um ambiente ao qual não foi exposto no treinamento do modelo. Essa questão também acontece na aprendizagem por reforço.

A aprendizagem profunda, por sua vez, requer grande quantidade de dados de treinamento previamente categorizados. Isso faz com que a sua utilização pura também não seja uma opção muito aplicada nesse tipo de problema.

Contudo, observa-se vantagens na aplicação de princípios das duas abordagens de aprendizagem, quando realizadas juntas. Dessa forma, a aprendizagem por reforço profundo é desenvolvida para melhorar a resolução desse tipo de problema, fazendo com que o agente aprenda durante o treinamento mas performe efetivamente em ambientes que não fizeram parte do treinamento.

Sendo assim, acredita-se que a aprendizagem por reforço profundo apresente vantagens no desenvolvimento de estratégias para navegação de robôs móveis frente a outras técnicas. A navegação sem mapeamento do ambiente é a base para muitos problemas em robótica móvel, onde muitos algoritmos de aprendizagem por reforço profundo têm, de fato, sido empregados. Essas técnicas têm alcançado um desempenho de ponta em alguns problemas de aprendizado de robôs, dada a evolução das redes neurais de aprendizado profundo.

Baseado nas informações mencionadas anteriormente, surge como tema proposto neste trabalho demonstrar e avaliar a eficácia de duas técnicas de aprendizagem por reforço profundo em tarefas que envolvem a navegação orientada a objetivos de um robô móvel terrestre com desvio de trajetória. São utilizados os algoritmos Deep Q-Network e Double Deep Q-Network, realizando uma comparação entre teste de simulação e real. Além disso,

os conhecimentos desenvolvidos neste projeto podem ser aplicados em amplas áreas da engenharia de controle e automação, para além da própria robótica.

Essa abrangência de aplicação de abordagens de aprendizagem por reforço profundo e o grande potencial de crescimento dessa área na indústria de tecnologia são relevantes não só cientificamente, mas também são a motivação pessoal para realização desse trabalho.

1.2 OBJETIVO GERAL E ESPECÍFICOS

O presente projeto tem como objetivo geral aplicar e comparar o desempenho de dois algoritmos de aprendizado por reforço profundo na navegação de um robô móvel com desvio de trajetória, em ambiente real e de simulação. Espera-se que o robô desvie obstáculos, atingindo um determinado ponto alvo sem mapeamento prévio do ambiente.

Visa-se também os seguintes objetivos específicos:

- Navegar com o robôs móvel Turtlebot 3 nos ambientes de simulação do Gazebo e em ambiente real;
- Treinar o algoritmo Deep Q-Network em ambientes de simulação;
- Treinar o algoritmo Double Deep Q-Network em ambientes de simulação;
- Construir os ambientes reais baseados nas simulações;
- Testar os modelos treinados em ambiente de simulação e real;
- Medir, apresentar e comparar os resultados obtidos durante os testes simulados e reais.

1.3 DIVISÃO DO TRABALHO

Este trabalho é dividido em cinco capítulos. O primeiro traz uma breve introdução, com justificativa e objetivos gerais e específicos. Na sequência, os trabalhos de outros pesquisadores da área que serviram de inspiração para este estão descritos no Capítulo 2, assim como fornece uma base teórica para os algoritmos usado nos experimentos. No Capítulo 3, são descritas as ferramentas, softwares e ambientes utilizados neste trabalho, assim como a metodologia usada para ensinar o agente a atingir um alvo, e a explicação da estrutura da rede e função de recompensa. Os resultados alcançados neste estudo estão

detalhados e discutidos no Capítulo 4. Por fim, a última seção traz uma conclusão com considerações finais sobre os objetivos atingidos e sugestão para trabalhos futuros.

2 REVISÃO DE LITERATURA

Neste capítulo serão apresentados alguns trabalhos relacionados e serão explicados alguns conceitos pertinentes ao projeto.

2.1 TRABALHOS RELACIONADOS

Existem alguns trabalhos relacionados a área do projeto. A aprendizagem por reforço profundo foi aplicada anteriormente a tarefas de navegação sem mapeamento para robôs móveis, em trabalhos como (CHEN et al., 2017), que desenvolve uma política de navegação eficiente para veículos robóticos navegarem com segurança e eficiência em ambientes com pedestres. O método proposto pelos autores permite a navegação totalmente autônoma de um veículo robótico movendo-se na velocidade de caminhada humana em um ambiente com muitos pedestres.

Em (JESUS et al., 2019), o autor opta por aplicar a técnica de aprendizagem por reforço profundo de Política do Gradiente determinístico Profunda para navegação do robô móvel e desvio de trajetória, utilizando como entrada a rede neural no treinamento 10 medições do sensor laser. A simulação é feita utilizando o software gazebo, e conclui confirmando a importância de um bom sistema de recompensas para atingir um bom aprendizado.

Também é importante citar (MNIH et al., 2013), que foi pioneiro em desenvolver um novo modelo, a época, de aprendizagem profunda aplicada a aprendizagem por reforço para jogar com sete jogos da Atari 2600. Os autores utilizam redes neurais na aplicação do algoritmo de Q-learning. Assim, desenvolvem a técnica de Deep Q-Learning, também referenciada como Deep Q-Network. Com as políticas aprendidas foi possível superar os jogadores humanos em três jogos e também os outros algoritmos usados em seis dos sete jogos.

É necessário destacar que o DQN apenas utiliza ações discretas quando aplicado a um problema como o controle de um robô. Para estender o DQN a um controle contínuo, (LILLICRAP et al., 2015) propôs um algoritmo de gradientes de política determinísticos profundos (DDPG). Este algoritmo abre caminho para o uso do Deep-RL na navegação de robôs móveis.

Em (DOBREVSKI; SKOCAJ, 2018), os autores adotaram uma abordagem para navegação orientada a objetivos sem mapeamento otimizando uma política comportamental dentro da estrutura de aprendizado por reforço, especificamente o método *Advantage Actor-Critic (A2C)*. Os resultados são comparados com o desempenho do pacote de navegação padrão Turtlebot3 e verificou-se que a abordagem alcançou um desempenho mais

robusto.

Entre os trabalhos relacionados ao algoritmo DQN, usado no projeto, pode-se citar (TAI; LIU, 2016a) que busca elaborar uma estratégia de movimento desviando os obstáculos do ambiente sem mapeamento prévio, utilizando informações de profundidade captadas por um sensor RGB-D. Ele aplica a técnica de aprendizagem por reforço utilizando o algoritmo de Deep Q-Learning com o robô, que possui o sensor, em diferentes ambientes de treinamento. Os resultados de simulação no Gazebo apresentaram robustez em diferentes ambientes.

Em (TAI; LIU, 2016b), o mesmo autor continua o estudo e realiza a comparação entre o treinamento da rede DQN e o treinamento utilizando apenas aprendizagem profunda, com redes neurais convolucionais, e também compara com o treinamento com apenas aprendizagem por reforço, com Q-Network. Para a comparação das redes, são realizados treinamentos em ambientes de simulação e realizados testes em ambientes reais e em ambientes de simulação. Ele conclui que o treinamento sob método de aprendizagem por reforço profundo se prova mais eficiente mesmo quando treinado somente em simulação e testado em ambiente real.

Em (TAI; PAOLO; LIU, 2017), Tai et al. aplica novamente aprendizagem por reforço profundo através de Deep Q-Learning para desenvolver estratégias de desvio de trajetória de robôs móveis em navegação. Tai et al. desenvolveu um planejador de movimento sem mapeamento para um robô móvel e usou como entrada para seu sistema 10 leituras de um sensor de alcance e a posição do alvo. E a saída da rede são os comandos de direção contínua para o robô. Inicialmente, eles empregaram comandos de direção discretos em (TAI; LIU, 2016b). Foi demonstrado que, com o método Deep-RL assíncrono, é possível treinar um agente para chegar a um alvo predeterminado usando um planejador de movimento sem mapear o ambiente.

Em (HASSELT; GUEZ; SILVER, 2016), os autores propõem então uma adaptação ao algoritmo Deep Q-Network. Eles mostram que o DQN sofre de superestimações substanciais em alguns jogos no domínio Atari 2600. O trabalho mostra que o algoritmo resultante, nomeado *Double Deep Q-Network (Double DQN)* não apenas reduz as superestimações observadas, mas também leva a resultados muito melhores de desempenho em vários jogos.

Assim como no trabalho de *Tai et al* em (TAI; PAOLO; LIU, 2017) e outros relacionados, este artigo se concentra no desenvolvimento de um planejador de movimento sem mapeamento do ambiente baseado em leituras de alcance de baixa dimensão. Esse trabalho se diferencia por utilizar uma abordagem determinística baseada no DDQN para a resolução de problemas relacionados à navegação de robôs móveis terrestres e a inserção de um alvo dinâmico para o robô nos ambientes sem treinamento assíncrono. No geral, mostramos que essas tarefas para robôs móveis terrestres podem ser superadas usando dados de sensoriamento de baixa dimensão e abordagens simples de Deep-RL, como o

Double DQN. Dessa forma, mostramos que um problema comum em Deep-RL, como a convergência do gradiente descendente ou os problemas de esquecimento catastrófico, podem ser consistentemente suavizados.

2.2 APRENDIZAGEM PROFUNDA

A aprendizagem profunda surgiu pela necessidade de aprimorar a área de aprendizagem de máquina para resolver problemas mais complexos. Com o avanço da capacidade de processamento das máquinas foi possível fazer isso de maneira mais rápida, resolvendo problemas que envolvam uma grande quantia de dados. Esse ramo da aprendizagem de máquina é baseada em redes neurais artificiais, que são inspiradas nas redes neurais do cérebro humano, simulando neurônios que possuem conexões, repassam e tratam informações.

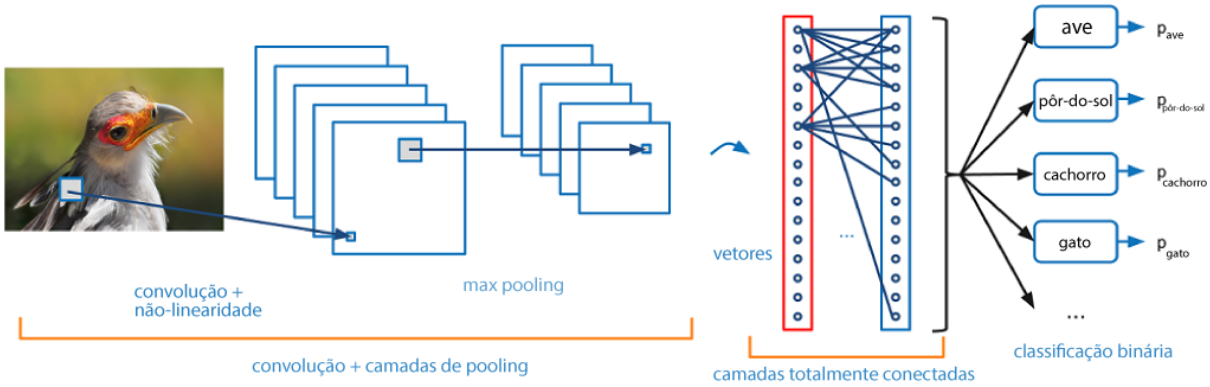
As redes neurais artificiais são tipicamente organizadas em camadas. Essas camadas podem possuir funções diferentes, e organizando-as de forma sequencial ainda pode-se obter diferentes resultados dependendo da ordem desta sequência. Diferentes arquiteturas podem ser empregadas na construção de uma rede neural.

Um exemplo de modelo de arquitetura de redes neurais muito comum são as redes neurais convolucionais, frequentemente referenciadas como ConvNet ou CNN - *Convolutional Neural Network*. A arquitetura de CNN é bastante utilizada principalmente em algoritmos que resolvem tarefas visuais complexas, envolvendo visão computacional, apesar de não ser restrita a essa aplicação. Essas redes podem ser utilizadas em outras aplicações como para reconhecimento de voz ou processamento de linguagem natural, por exemplo.

As redes neurais convolucionais possuem camada de entrada, camadas ocultas e camadas de saída. A ideia de convolução aparece para a identificação de padrões entre as entradas da rede. Dessa forma é possível encontrar similaridades como bordas, podendo ser utilizada como uma espécie de filtro.

Um exemplo de problema de classificação é trazido na Figura 1. No exemplo a entrada é um pixel da imagem, e na saída temos a probabilidade calculada pela rede neural da imagem ser um pássaro, um pôr-do-sol, um gato ou um cachorro. Tipicamente, para resolver um problema de classificação se utiliza em adição a CNN uma camada Totalmente Conectada.

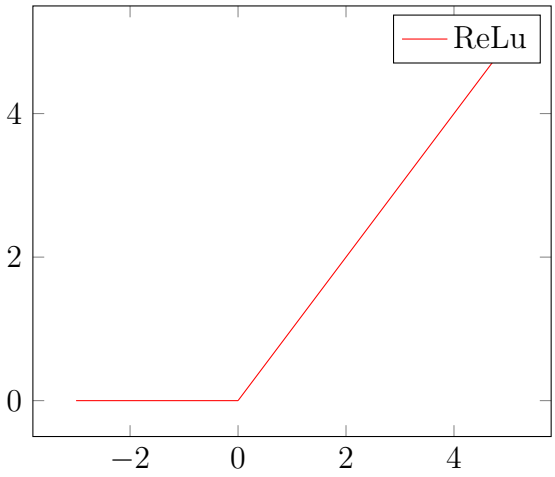
Figura 1 – Exemplo de Redes Neurais Convolucionais num problema de classificação



Fonte: Traduzido de https://analyticsindiamag.com/comprehensive-guide-to-different-pooling-layers-in-deep-learning/

As redes neurais convolucionais utilizam tipicamente uma função ReLU para ativação. A função ReLU - ou *Rectified Linear Unit*- é um retificador linear e seu comportamento é mostrado no gráfico da Figura 2. Como pode-se observar, ela tem saída zero para valores negativos e não satura na região positiva. Também converge mais rápido do que outras funções de ativação, como a sigmoide ou a tangente hiperbólica, por isso apresenta bons resultados para processamento de imagens.

Figura 2 – Função de ativação ReLU



Fonte: Autor.

As camadas totalmente conectadas realizam a classificação na saída da rede. A última camada geralmente é do tipo *Dense*, e deve possuir o número de neurônios igual ao número de saídas da rede, ou seja, o mesmo número de classes possíveis pro problema de classificação em questão.

2.3 APRENDIZAGEM POR REFORÇO

A aprendizagem por reforço é uma técnica da aprendizagem de máquina. O seu princípio básico é que o agente interaja com o ambiente e tome decisões através de um sistema de recompensas, em que ele tenta realizar as ações de forma a maximizar suas recompensas ao longo do tempo.

Para melhor entendimento desta técnica, é necessário revisar alguns conceitos. O agente é como é chamada a entidade que toma decisões no ambiente, é quem aprende no sistema. O ambiente é o espaço em que o agente realiza suas ações, sendo real ou em uma simulação. Estado é como esses elementos se encontram naquele momento, é como o sistema está, de forma que quando o agente toma uma decisão e faz uma ação, o estado do sistema muda.

Para o agente, no início e durante a ação que ele está executando não há resposta certa. Primeiro ele é posto no ambiente pra interagir e então aprende de acordo com as recompensas que obtém ao se locomover, as quais podem ser positivas ou negativas. As recompensas negativas, que também podem ser chamadas de punições ou penalidades, são relacionadas a ações a serem evitadas, e recompensas positivas são utilizadas como reforço positivo para as ações que se deseja aproximar, ações que levem o agente para mais próximo a seu objetivo de maneira menos custosa. Dessa forma, pode-se dizer que o agente aprende por tentativa e erro, sendo esperado que ao final do treinamento ele atinja seu objetivo com maior eficácia.

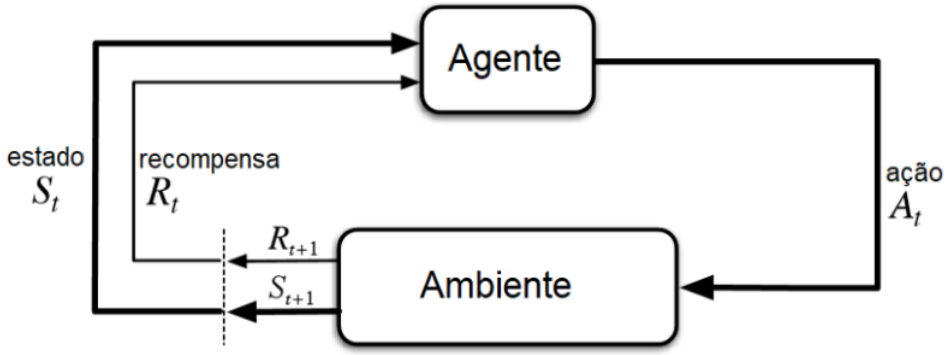
Um episódio é normalmente definido como conjunto de estados que se passam entre um estado inicial e um estado terminal. Em um algoritmo, os estados terminais podem ser de três tipos: colisão, atingir objetivo ou *time out*. A colisão é quando o robô colide nas paredes do ambiente ou em obstáculos. Atingir o objetivo é quando o robô passa por cima do local definido como objetivo naquele episódio. *time out* é quando o tempo máximo estipulado pra um episódio é atingido e o agente não alcança nenhum estado terminal.

A lógica básica da aprendizagem por reforço é um circuito entre agente, ação e estado. Nesse processo, o robô é considerado o agente, a localização dele é considerada como sendo o estado e o movimento dele, ou processo de decisão, é a ação. A política, nesse contexto, é a função que mapeia a probabilidade do agente executar cada ação em cada estado possível, sendo uma estratégia que o agente pode ter para obter o melhor resultado. Quando um algoritmo é orientado por política, diz-que que ele é *on policy* e quando ele não é orientado por política utiliza-se *off policy*.

Essa técnica é geralmente modelada matematicamente utilizando o Processo de Decisão de Markov mostrado na Figura 3. Na estrutura, citada por (SUTTON; BARTO, 2018), o agente toma uma ação A no instante t e o ambiente retorna um novo estado S e uma nova recompensa R para o instante t+1. O processo acontece novamente de forma

iterativa.

Figura 3 – Estrutura do Processo de Decisão de Markov



Fonte: Traduzido de (SUTTON; BARTO, 2018, p. 54)

Pode-se perceber que essa modelagem depende apenas do estado e ação do próximo instante e não depende das ações e estados anteriores. Essa é uma condição para que o sistema atenda a propriedade de Markov. Dessa forma:

$$P(s_{t+1}|s_0, a_0, s_1, a_1, ..., s_t, a_t) = P(s_{t+1}|s_t, a_t) \tag{2.1}$$

Na equação, P é a probabilidade calculada pela decisão de Markov.

2.4 APRENDIZAGEM POR REFORÇO PROFUNDO

Como apresentado anteriormente, a aprendizagem profunda é um campo de estudo que tem crescido nos últimos anos. Dentro dele, a técnica de criação de redes neurais artificiais é utilizada em diversas aplicações, e a robótica é uma delas. A técnica de aprendizagem por reforço profundo alia as redes neurais ao conceito de aprendizagem por reforço, com o objetivo de resolver problemas que não demonstram resultados satisfatórios aplicando somente uma das técnicas.

Existem diversos modelos de algoritmos, como SARSA, Deep Q-Network ou Política de Gradiente Determinística Profunda. Um fator que pode influenciar na escolha do algoritmo utilizado é se espaço de estados do problema é em tempo contínuo ou discreto. Alguns são adequados para o espaço de tempo continuo e outros apenas para o discreto. O processo de Decisão de Markov, citado anteriormente, geralmente modela sistemas com ações e estados de tempo discreto.

O SARSA é uma modificação do algoritmo de Q-Learning, como é demonstrado por (XU et al., 2018). Enquanto o Q-Learning é baseado em valor, o SARSA é um algoritmo baseado em política. Ele utiliza a ação e estado atual para calcular o valor de

Q, diferentemente do Q-learning original, que utiliza apenas o estado do instante futuro. O SARSA é um algoritmo que evita o alto risco, sendo assim considerado mais conservador, e isso pode fazer com o que o aprendizado seja mais demorado.

Tanto nas técnicas de aprendizagem por reforço quanto nas de aprendizagem por reforço profundo, o agente não aprende com apenas um episódio de treinamento. São necessários múltiplos episódios para avaliar a eficiência do método aplicado. A quantidade ideal de episódios varia de acordo com cada algoritmo, agente, ambiente e ação de cada sistema.

2.5 ALGORITMO DE Q-LEARNING

O Q-Learning é um tipo de algoritmo de aprendizagem por reforço. Este algoritmo é considerado off policy, ou não orientado por política, mas sim baseado em valor. Nele, é determinado valor de Q para cada par de estado-ação (s,a), de maneira iterativa, para gerar uma política que maximize o total de recompensas recebidas depois de todos os estados. O Q-learning também é um algoritmo para um espaço de ações discreto, mas há a possibilidade de discretizar estados e ações quando aplicada para sistemas em tempo contínuo. Para compreender a dedução até o algoritmo de Q-Learning, será utilizada nessa seção o caminho apresentado por (GERON, 2019).

Antes de compreender a equação que determina o valor de Q, é preciso revisar a Equação de Otimização de Bellman. Através da equação 2.2, Bellman estima o valor ideal de um estado s, posto $V^*(s)$, que é a soma de todas as recompensas futuras descontadas que o agente pode esperar, em média, após atingir um estado s, supondo que ele atue de forma otimizada. Na equação, γ é a taxa de desconto, $T(s, a, s')$ é a probabilidade de transição do estado s para o estado s' quando o agente faz a ação a, e $R(s, a, s')$ é a recompensa que o agente obtém quando passa de um estado s para um estado s' efetuando a ação a.

$$V^*(s) \;=\; \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma . V^*(s')] \tag{2.2}$$

Em resumo, essa equação recursiva diz que, se o agente age de forma ideal, o valor ideal do estado atual é igual à recompensa que ele obterá, em média, após tomar uma ação otimizada mais o valor ideal esperado de todos os próximos estados possíveis que essa ação possa levar.

Apesar de sua importância, essa equação ainda não se referia a ação que deveria ser tomada. Para essa definição ser mais direta e útil para aplicações práticas, Bellman adapta sua equação. Agora, o Algoritmo de Iteração de Valor de Q conseguiria estimar qual é a melhor ação a ser tomada em determinado estado, bastando inicializar as estimativas dos

valores de Q em zero. A representação matemática de tal algoritmo é definida por:

$$Q_{(k+1)}(s,a) \leftarrow \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma . \max_{a} Q_k(s',a')] \tag{2.3}$$

Como mencionado, essa equação exige a definição de um valor inicial do valor de Q. O algoritmo final de Q-Learning é uma adaptação do Algoritmo de Iteração do Valor de Q para a situação em que as probabilidades de transição e as recompensas são inicialmente desconhecidas. O algoritmo de Q-Learning pode ser expresso matematicamente pela equação:

$$Q_{(k+1)}(s,a) \leftarrow (1-\alpha)Q_k(s,a) + \alpha(r + \gamma . \max_{a'} Q_k(s',a')) \tag{2.4}$$

Após cada iteração do algoritmo, o valor de Q é atualizado para o mais otimizado, ou seja, o que obtém a maior soma de recompensas após o episódio. O valor "Q"está para "qualidade", e pode-se interpretar que o seu valor represente o quão útil aquela ação é para obter a melhor recompensa, ou a sua qualidade em relação a obter a maior recompensa. Uma forma diferente de representar o algoritmo para melhor visualização é descrita a seguir:

Algoritmo 1: Q Learning

Entrada: Dados taxa de aprendizagem $\alpha \epsilon [0,1], \epsilon\ \varepsilon [0,1] Pequeno$

Saída: $\pi \approx \pi *$

Inicializar Q(s, a) arbitrariamente para todos estados e ações, exceto para o estado terminal onde Q = 0

para cada *episódio* **faça**

  Inicialize s

  para cada *passo do episódio* **faça**

    Escolha a no estado S usando política derivada de Q;

    Tome a ação a, observe r, s'

    Q(s,a) $\leftarrow\ Q(s,a) + \alpha(r + \gamma . \max_a Q(s',a) - Q(s,a))$

    s $\leftarrow\ s'$

  fim

fim

Dessa forma, pode-se dizer que com essa equação o algoritmo de aprendizagem por reforço consegue encontrar o maior valor de Q para determinado estado e ação.

2.6 DEEP Q-LEARNING

No Q-Learning, o aumento da quantidade de estados e ações pode tornar o algoritmo mais complexo, de forma que fique muito demorado e custoso. Para melhorar o desempenho do Q-Learning, surge a Rede Q Profunda - ou Deep Q-Learning. Aqui será chamada de DQN, em referência a Deep Q-Network. A DQN é um aperfeiçoamento do algoritmos do tipo Q-Learning, aliando este conceito a conceitos de aprendizagem profunda, ajudando a resolver os problemas mais custosos. Essa abordagem foi primeiro desenvolvida por (MNIH et al., 2013), trabalho citado na seção de trabalhos relacionados desse trabalho.

Dentro da aprendizagem profunda, a DQN utiliza redes neurais convolucionais em sua aplicação, para aproximar a função de valor de ação ótima. Assim, é possível estimar futuras recompensas, sem depender puramente das tentativas e obtenção de punição e recompensas para a aprendizagem. Utilizando as redes neurais, obtém-se um valor de Q futuro aproximado para cada ação.

Então seleciona-se o maior e é descontado para obter uma estimativa do valor descontado futuro. Somando a recompensa r e o valor descontado futuro estimado, obtém-se o valor de Q alvo para o par estado-ação (s,a). Com o valor de Q desejado, executa-se uma iteração de treinamento utilizando qualquer algoritmo de Gradiente Descendente, que busca minimizar o erro quadrático entre o valor de Q estimado e o Valor de Q desejado. Esse é o procedimento básico do DQN.

Esta rede neural pode ser treinada minimizando uma perda $\mathcal{L}(\theta)$ dada pela seguinte equação:

$$\mathcal{L}(\theta) = [(y - Q(s, a, \theta))^2]. \tag{2.5}$$

Com y sendo uma função alvo representada pela Equação 2.6, empregando os pesos de rede θ do episódio t, usada para criar uma diferença temporal entre o valor encontrado e o valor esperado.

$$y = \mathbb{E}_{s' \sim \varepsilon}[r + \gamma \max_{a'} Q(s', a'; \theta_t) | s, a] \tag{2.6}$$

As entradas para a rede neural são um par estado-ação (s,a) amostrado aleatoriamente de um *buffer* de repetição de experiência D, que armazena cada transição alcançada pelo agente. Isso torna o método *model-free* - sem modelo, reduzindo os problemas de dados correlacionados e distribuições não estacionárias. A ação escolhida pelo agente segue uma política gulosa (ϵ-greedy), com $a = \underset{a}{max} Q(s, a; \theta)$ quando $1 - \epsilon$ for *true* ou uma ação aleatória com probabilidade ϵ.

O algoritmo, de acordo com o demonstrado por (MNIH et al., 2013) e (MNIH et al., 2015), pode ser visualizado no pseudocódigo a seguir:

Algoritmo 2: Deep Q-Learning com Experiência de repetição

Inicializar memória de repetição D.

Inicializar função-ação Q com pesos aleatórios.

para cada *episódio* **faça**

  Inicialize s.

  para cada *passo do episódio* **faça**

    Com probabilidade ϵ selecione uma ação aleatória a

    senão, selecione $a = \max_a Q^*(\text{s,a};\theta)$

    Execute ação a e observar a recompensa r e estado s'

    Armazene transição (s, a, r, s') em D

    Defina $s \leftarrow s'$

    Retire um *minibatch* de amostra aleatório das transições(s, a, r, s') de D

    Defina $y_j = \begin{cases} r \text{ para passos terminais} \\ r + \gamma \max\limits_{a'} Q(s', a'; \theta) \text{ para passos não-terminais} \end{cases}$

    Calcule a perda $\mathcal{L}(\theta)$ conforme equação 2.5

    Performe o passo do gradiente descendente

  fim

fim

2.7 ALGORITMO DE DOUBLE DEEP Q-LEARNING

Nas redes Q-Learning simples e Deep Q-Network, que foram apresentadas no tópico anterior, o operador *max* emprega os mesmos valores para selecionar e avaliar uma ação. Isso aumenta a probabilidade de escolher valores superestimados, podendo levar a estimativas de valor excessivamente otimistas.

Como uma possível solução para o problema de valores superestimados, (HASSELT, 2010) desenvolveu um novo algoritmo chamado Double Q-learning, que usa uma abordagem de estimador duplo para estimar o valor do próximo estado. Basicamente, com estimador duplo, o Double Q-learning armazena duas funções Q. Tem-se também dois conjuntos de pesos, θ e $\theta^{'}$. Cada função Q é atualizada com um valor da outra função Q para o próximo estado. Para uma comparação clara com o Q-Learning, a função alvo no Double Q-learning pode ser reescrita como:

$$y_t^{DoubleQ} = r' + \gamma Q(s', \underset{a}{argmax} Q(s', a; \theta); \theta_t) \tag{2.7}$$

O erro Double Q-learning pode então ser escrito como

$$y_t^{DoubleQ} = r' + \gamma Q(s', \underset{a}{argmax} Q(s', a; \theta); \theta_t^{'}) \tag{2.8}$$

Observa-se que a seleção da ação, no *argmax*, é ainda devido aos pesos online θ. Isso significa que, assim como no Q-learning, ainda estamos estimando o valor da política *greedy* (gulosa) de acordo com os valores atuais, conforme definido por θ. No entanto, usamos o segundo conjunto de pesos $\theta_t^{'}$ para avaliar de forma justa o valor desta política. Este segundo conjunto de pesos pode ser atualizado simetricamente trocando os papéis de θ e $\theta^{'}$.

Utilizando a estratégia Double Q-Learning associada ao Deep Q-Network, (HASSELT; GUEZ; SILVER, 2016) trazem uma nova abordagem chamada Double Deep-Q Network (DDQN). Foi proposto, avaliar a política gulosa de acordo com a rede online, mas usando a rede alvo (target) para estimar seu valor. Sua atualização do valor é a semelhante a do DQN, mas substituindo função alvo.

No DDQN, a função alvo é apresentada como:

$$y_t^{DoubleDQN} = r' + \gamma Q(s', \underset{a}{argmax} Q(s', a; \theta), \theta_t^-) \tag{2.9}$$

onde (θ_t^-) é o peso da rede alvo para a avaliação da política *greedy* atual. Em comparação com o Double Q-learning, os pesos do rede $\theta_t^{'}$ são substituídos pelos pesos da rede alvo θ_t^- para a avaliação da atual política gulosa. A atualização para a rede alvo permanece inalterada em relação ao DQN e permanece uma cópia periódica da rede online. O algoritmo *Double DQN* é apresentado a seguir:

Algoritmo 3: Double Deep Q-Learning

```
Inicializar memória de repetição D
Inicializar as duas redes Q com pesos aleatórios
para cada episódio faça
    Inicialize s
    para cada passo do episódio faça
        Observe o estado s e com probabilidade ϵ selecione uma ação aleatória a
        senão, selecione a = max_a Q*(s,a;θ)
        Execute ação a e observe a recompensa r e estado s'
        Armazene transição (s, a, r, s') em D
        Defina s ← s'
        Retire um minibatch de amostra aleatório das transições(s, a, r, s') de D
        Calcule a função alvo y_t^DoubleDQN de acordo com a equação 2.9
        Calcule a perda L^DDQN(θ).
        Performe o passo do gradiente descendente
    fim
fim
```

O cálculo $\mathcal{L}^{DDQN}(\theta)$ da perda segue a equação 2.5, mesma utilizada no DQN, mas, nesse caso, utilizando a função alvo do $y_t^{DoubleDQN}$ no lugar de y.

3 MATERIAIS E MÉTODOS

Neste capítulo, serão descritos os materiais e métodos utilizados no projeto. O trabalho é desenvolvido em uma imagem do Docker com sistema Ubuntu 20.04, utilizando o ROS Noetic. O robô escolhido para simulação é um Turtlebot3.

3.1 DOCKER

Docker é uma plataforma de código aberto que facilita a criação e administração de códigos isolados. Nele é possível possuir uma parte de um sistema operacional em seu computador, semelhante a uma máquina virtual mas sem precisar de todo o sistema operacional.

Nesse caso, temos uma imagem de um *container* do Docker com Ubuntu 20.04 e ROS Noetic instalado, assim como Pytorch e todas os frameworks e bibliotecas necessárias para o funcionamento dos códigos do robô.

3.2 PYTORCH

Python foi a linguagem de programação utilizada no desenvolvimento dos algoritmos, e possui aplicações em diversas áreas, como processamento de imagens(PFITSCHER et al., 2019) e robótica (SILVA; CUADROS; GAMARRA, 2020). Uma das bibliotecas do Python é o PyTorch.

PyTorch é uma biblioteca de aprendizado de máquina *open-source* aplicada aqui na criação de redes neurais profundas (SUBRAMANIAN, 2018). Ele fornece dois recursos de alto nível: computação tensorial com aceleração por meio de uma unidade de processamento gráfico e um sistema de diferenciação automática baseado em fita. É muito apreciado por sua facilidade de uso e simplicidade e incorpora conceitos do Python, como classes, estruturas e loops condicionais.

Essa biblioteca se popularizou quando indicadores de desempenho e agilidade se tornaram essenciais para o desenvolvimento. A escalabilidade do PyTorch está ligada à facilidade e eficiência de uso e paralelismo e aceleração de hardware. As aplicações comerciais do Pytorch estão crescendo rapidamente, com empresas como Tesla, Facebook (WU et al., 2019), Uber e muitas outras utilizando-o. No meio acadêmico já é bastante utilizado em pesquisas em áreas como processamento de linguagem natural, processamento de imagens, reconhecimento de objetos, entre outros (LASKIN; SRINIVAS; ABBEEL, 2020), (DAI et al., 2019), (RAO; MCMAHAN, 2019).

3.3 ROBOT OPERATIONAL SYSTEM

O ROS - *Robot Operational System* - é um conjunto de pacotes de *software open-source* utilizado para aplicações de robótica. Lançado em 2007, este foi desenvolvido para ajudar pesquisadores e desenvolvedores a utilizar sistemas robóticos.

É importante ressaltar que, apesar de ter um conjunto de pacotes padrão, o ROS é um framework flexível e bastante adaptável a diversas aplicações robóticas. O objetivo é que a partir da utilização do ROS o pesquisador consiga evitar o processo de desenvolver desde o início esses pacotes básicos para começar um projeto de robótica, o que antes tornaria o projeto mais demorado e complexo.

Também, por possuir esse caráter de colaboração principalmente no meio acadêmico, o ROS possui uma grande comunidade de pesquisadores que disponibilizam repositórios de soluções em robótica. Essa soluções são úteis para adiantar pequenas questões de projeto, como os ambiente simulados nesse projeto, que será apresentado em **??**.

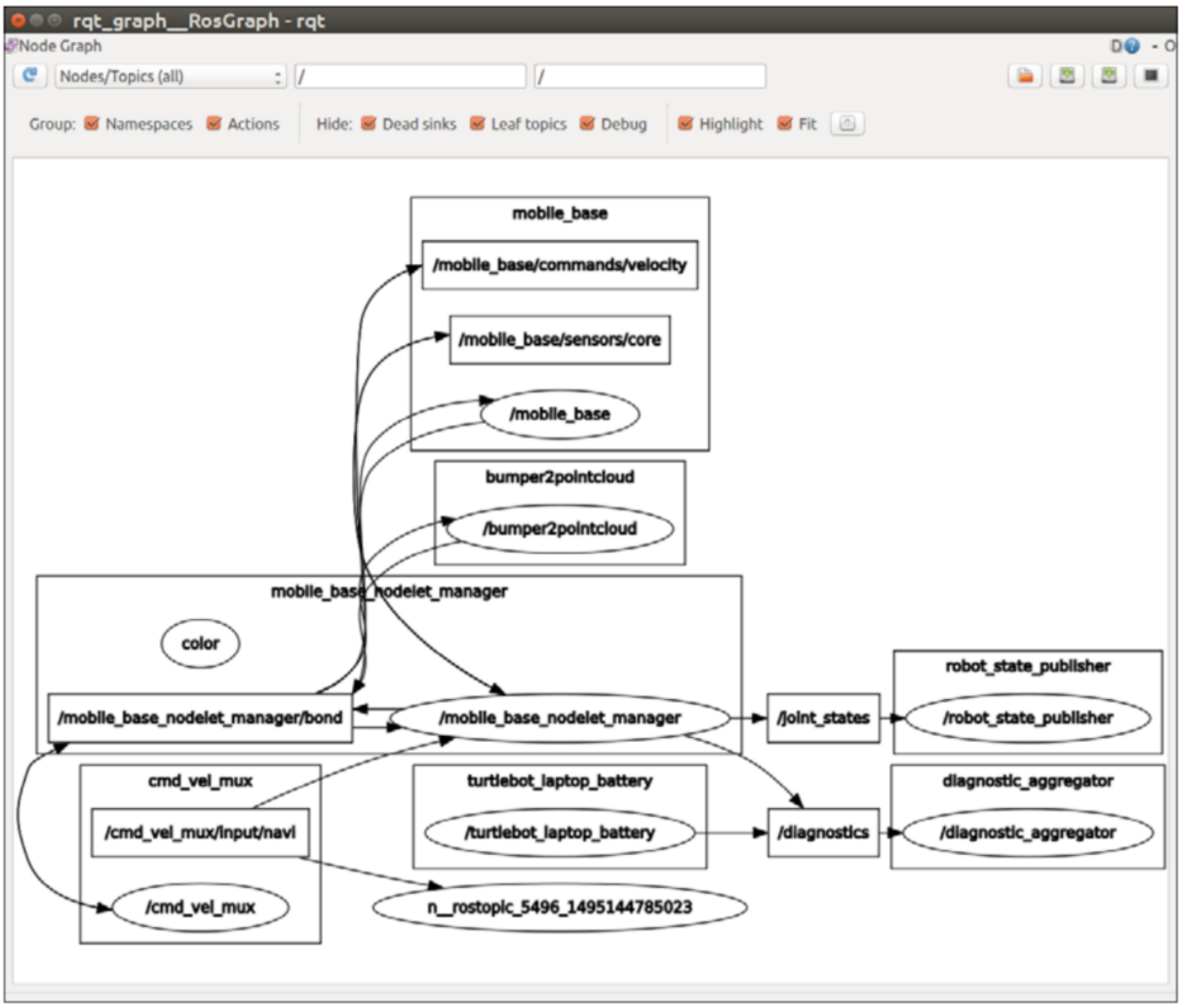
Esse sistema opera em plataformas baseadas em Unix, apresenta atualizações regulares e novas versões. As versões são comumente associadas a uma versão de sistema operacional do computador, e as informações são disponibilizadas no site oficial do ROS (Stanford Artificial Intelligence Laboratory et al.,). O ROS Noetic é a versão utilizada nesse projeto e é recomendada para Ubuntu 20.04.

Os pacotes de ROS se organizam em uma estrutura de nós, tópicos e publicação e leitura de mensagens. A publicação de mensagens é basicamente a escrita de um dado em um tópico, como por exemplo, um sensor de posição enviando os dados das coordenadas xyz ao tópico de odometria do robô. Para a leitura de um dado, o nó deve se inscrever naquele tópico, assim poderá acessar os dados que o tópico receber.

O gráfico da estrutura de nós pode ser visualizado de forma simples utilizando o recurso *rqt graph*, nativo do ROS, que mostra os nós e tópicos que estão ativos quando uma simulação está sendo processada. Na Figura 4, é apresentado um exemplo de gráfico de nós de uma simulação simples do TurtleBot2 no Gazebo, encontrada no livro (FAIRCHILD; HARMAN, 2017). Os nomes nos retângulos representam os tópicos, os nomes nas elipses representam os nós, e as setas são as conexões entre os tópicos. Essas setas mostram a direção do fluxo de dados, seja através da publicação ou do recebimento de mensagens. No exemplo, o nó */mobile_base_ nodelet_manager* publica nos tópicos */joint_states* and */diagnostics*.

Na Figura 5, há um recorte para facilitar a compreensão. Quando se roda o nó de */turtlebot_teleop_keyoard*, esse nó aparece no fluxograma, publicando no tópico */cmd_vel_mux/input/teleop*. Este nó */turtlebot_teleop_keyoard* tem a funcção de Teleoperação do robô, ou seja, possibilita a operação da navegação do turtlebot através do teclado. Na prática, o tópico está enviando as informações de quais teclas estão sendo pressionadas no teclado, ou seja, qual comando de velocidade angular está sendo enviada

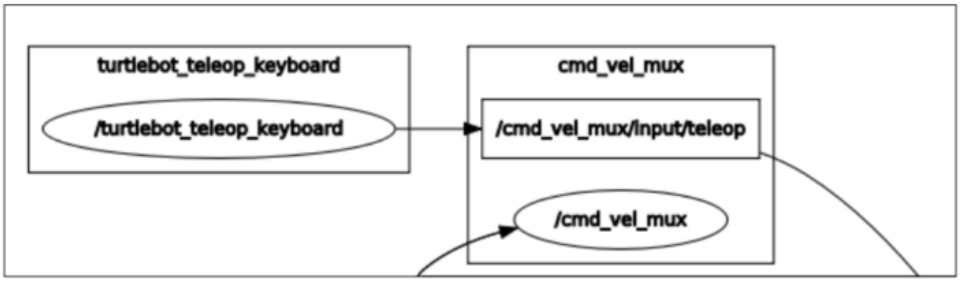
Figura 4 – Estrutura de nós do ROS



Fonte: Extraído de (FAIRCHILD; HARMAN, 2017, p. 105)

para que o robô navegue.

Figura 5 – Estrutura de nós ampliada do ROS com Teleoperação



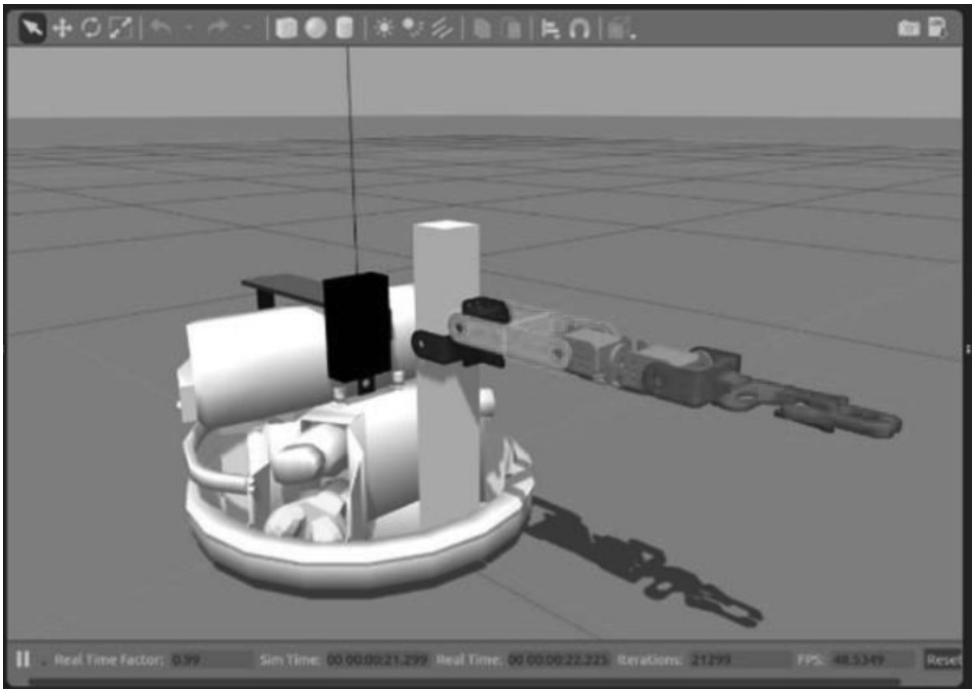
Fonte: Extraído de (FAIRCHILD; HARMAN, 2017, p. 106)

3.3.1 Gazebo

O gazebo é um software de simulação tridimensional do ROS. É possível realizar diversas simulações, incluindo robôs móveis e ambientes com os quais eles interagem. Por isso, ele é muito utilizado para simulações dos Turtlebots e foi escolhido para este projeto.

Na Figura 6 pode-se observar um robô manipulador sendo simulado no *software* Gazebo.

Figura 6 – Simulação de robô manipulador no Gazebo

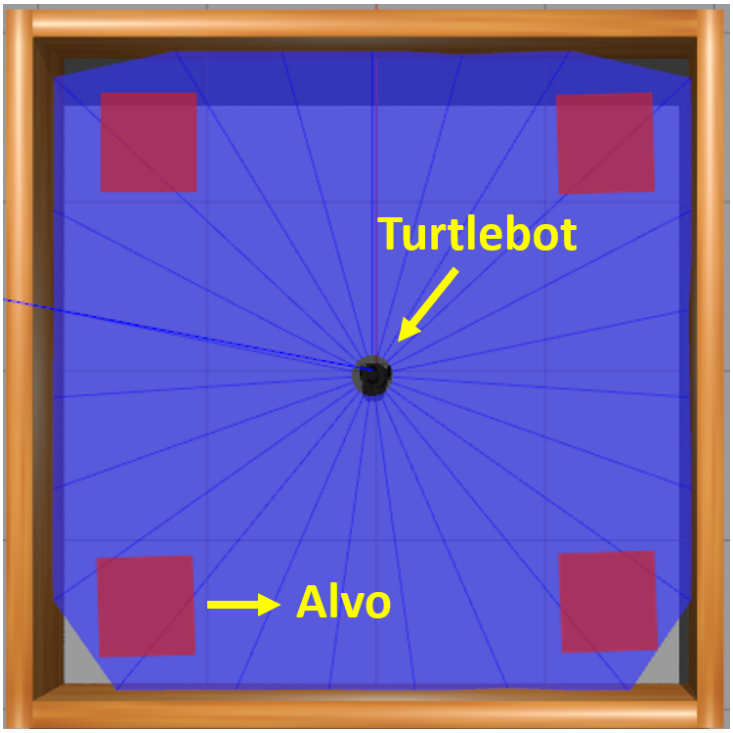


Fonte: Extraído de (ANGGRAENI; ROKHIM; SALAM, 2020)

3.3.2 Ambientes de simulação

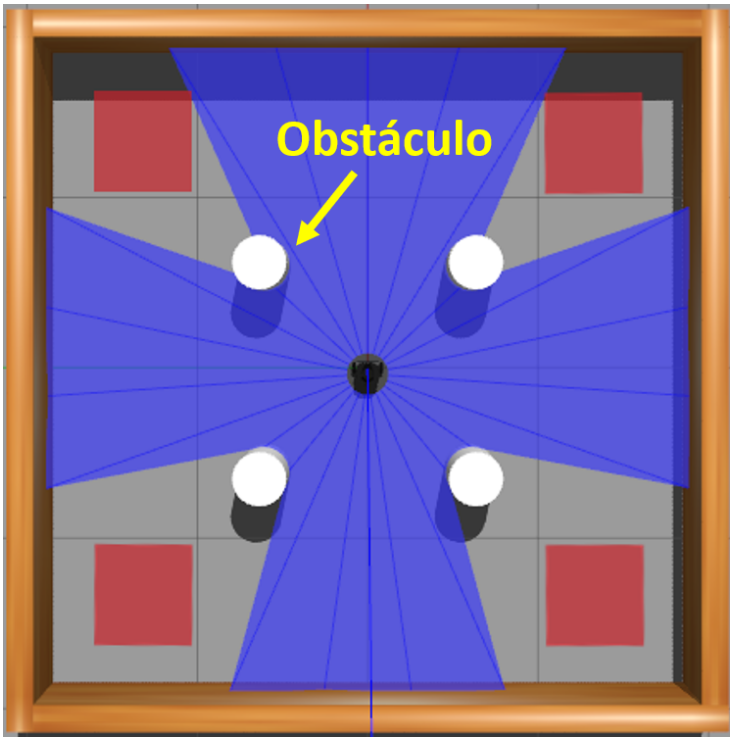
Os dois primeiros ambientes de simulação escolhidos no gazebo estão disponíveis no repositório da ROBOTIS. Dentre os dois, um é um ambiente fechado, como simulando um quarto quadrado só com quatro paredes, mostrado na Figura 7, e o outro utiliza esse mesmo quadrado e adiciona quatro cilindros como obstáculos, mostrado na Figura 8. O ambiente sem obstáculos é denominado Ambiente 1 e o com obstáculos, Ambiente 2.

Figura 7 – Ambiente de simulação 1



Fonte: Autor

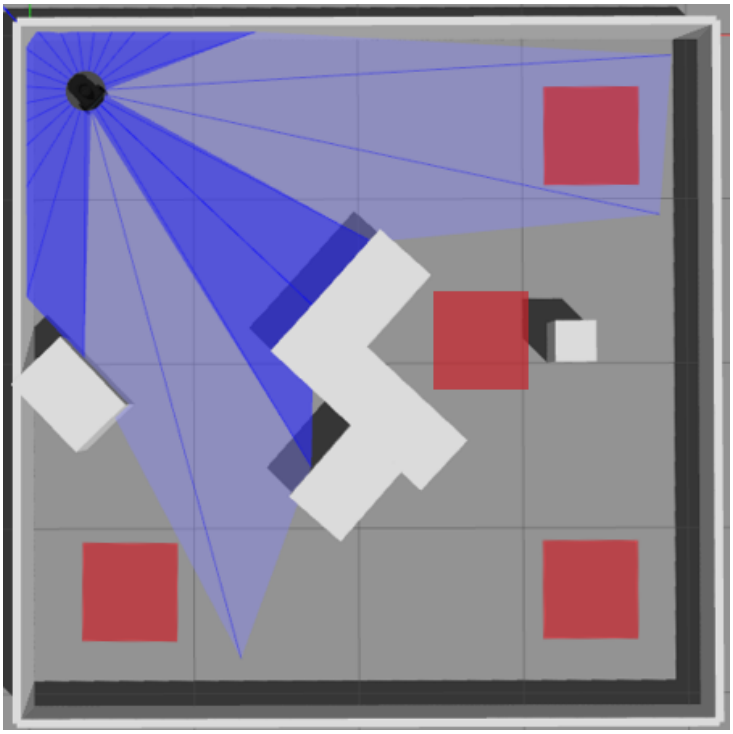
Figura 8 – Ambiente de simulação 2



Fonte: Autor

Dentro do ambiente de simulação, o robô tem como objetivo atingir o alvo sem colidir com as paredes e, no caso do ambiente 2, também sem colidir com os obstáculos. O alvo é um ponto estipulado no mapa, e na simulação é representado graficamente por um quadrado vermelho. Os alvos nas imagens correspondem aos fixados para teste do algoritmo com os modelos já treinados em simulação. O terceiro ambiente, nomeado ambiente 3 ou ambiente L, foi criado para simular obstáculos com maior semelhança aos reais, com caixas distribuídas pelo mapa. Na Figura 9 , é possível observar o Turtlebot3 no Ambiente 3, com a representação gráfica dos alvos fixados para realização dos testes.

Figura 9 – Ambiente de simulação 3



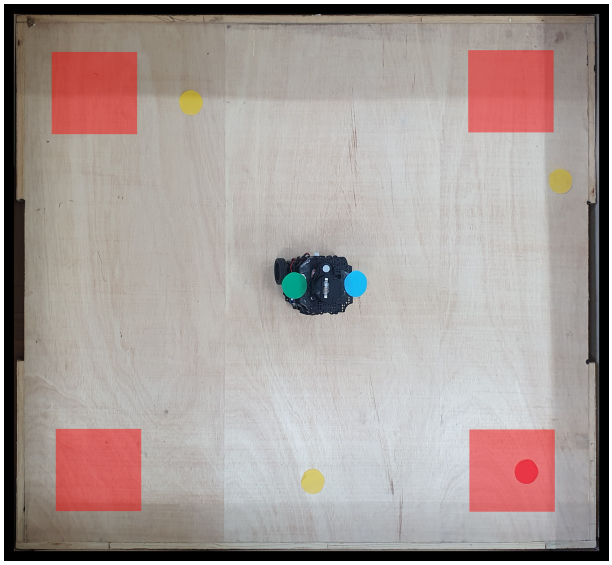
Fonte: Autor

3.4 AMBIENTES REAIS

Após o treinamento e teste das redes em ambiente simulado, são realizados os testes em ambiente real com o TurtleBot3. Para isso, são construídos os 3 ambientes em laboratório, com proporções semelhantes a de simulação, mas reduzido em 2.66 vezes, devido a limitação de espaço disponível na sala. O primeiro ambiente real é mostrado na Figura 10 e se assemelha à primeira simulação sem obstáculos. O segundo ambiente é mostrado na Figura 11, e aumenta um pouco o nível de dificuldade em relação ao primeiro cenário utilizado. O terceiro e último cenário é mostrado na Figura 12, e apresenta um

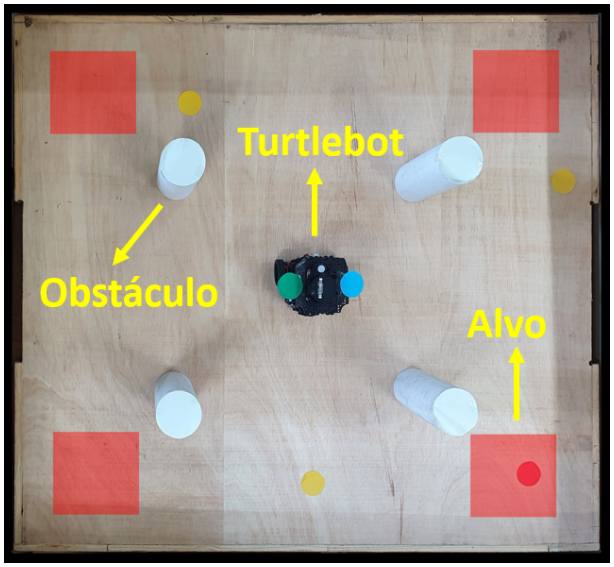
cenário complexo semelhante ao terceiro utilizado na simulação.

Figura 10 – Ambiente real 1



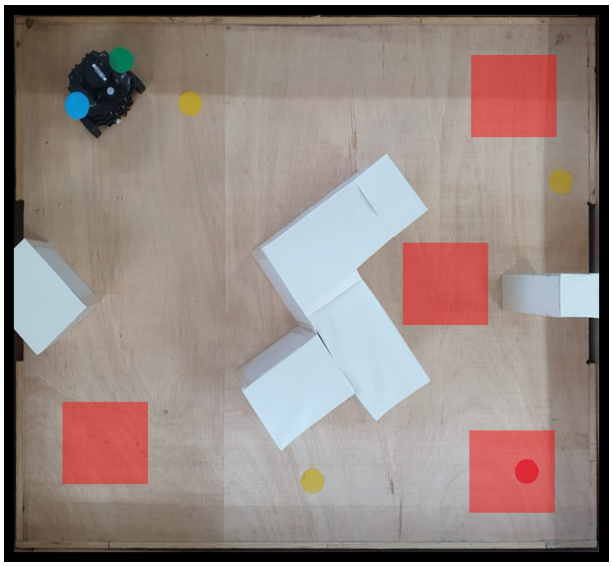
Fonte: Autor

Figura 11 – Ambiente real 2



Fonte: Autor

Figura 12 – Ambiente real 3



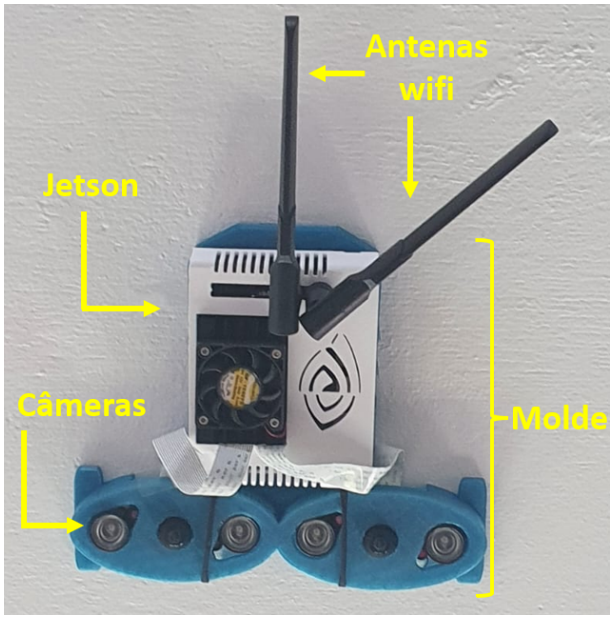
Fonte: Autor

Também, para os testes em ambiente real, câmeras com visão superior capturam imagens do cenário e executam algoritmos de processamento digital de imagens. A biblioteca de visão computacional OpenCV foi usada para processar essas imagens. OpenCV é a biblioteca mais utilizada para visão computacional, e também é uma biblioteca de código aberto.

3.4.1 Visão Computacional e OpenCV

Para a visão de topo, foi utilizado o sistema embarcado Jetson Nano B01. Foi desenvolvido um molde em impressão 3D para acoplar os componentes e fixar no teto do laboratório, e o resultado pode ser observado na Figura 13. A conexão entre os aparelhos (o Turtlebot, o notebook e a Jetson) é realizada através do ROS, na mesma rede compartilhando os mesmos grafos de nós.

Figura 13 – Jetson Nano B01 com periféricos



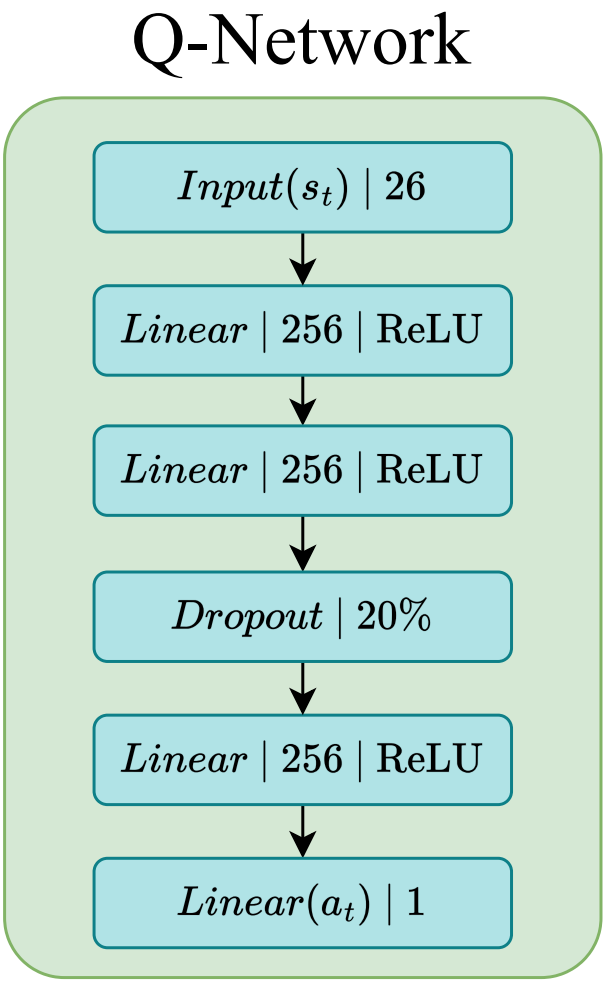
Fonte: Autor

Esta etapa de câmeras e visão do projeto não foi desenvolvido no trabalho atual, apenas aplicado. O algoritmo de visão computacional foi desenvolvido pelo grupo de pesquisa GARRA.

3.4.2 Estrutura da Rede Q

Após o estabelecimento dos estados e ações do sistema, foi criada uma rede Q para compor as arquiteturas DQN e Double DQN. A rede Q possui 26 entradas, incluindo as medições do sensor de alcance do laser, velocidades angulares e lineares anteriores e a posição e orientação do alvo. A estrutura de rede aplicada no DQN e DDQN possui 256 neurônios, 3 camadas totalmente conectadas e uma camada de *Dropout*, e é mostrada na Fig 14. A função de ativação utilizada é a *ReLU*.

Figura 14 – Rede Q utilizada



Fonte: Autor.

A saída da rede representa uma ação a ser tomada pelo robô. Essa ação é um valor discreto entre [0,4] e representa a sua velocidade angular. A relação entre o valor de saída da rede e a velocidade angular é mostrada na Tabela 1. A Ação 0 representa virar para a esquerda, 1 virar um pouco para a esquerda, 2 é seguir em frente, 3 é um pouco para a direita e 4 é virar para a esquerda.

Tabela 1 – Ação e velocidade angular

Ação	Velocidade Angular
0	$-1,5\ rad/s$
1	$-0,75\ rad/s$
2	$0\ rad/s$
3	$0,75\ rad/s$
4	$1,5\ rad/s$

3.4.3 Função de recompensa

Uma vez definidos os ambientes simulados, é possível simular o robô na tarefa de navegação. Primeiro, a rede Deep-RL precisa ter o mecanismo de recompensa e penalidade especificado. As recompensas e penalidades dadas ao agente são apenas números de uma função que modela como ele queria que o agente se comportasse. Pode ser baseado em conhecimento empírico e criado durante o processo de solução do problema. Em relação ao sistema de recompensas, existem três condições diferentes que apresentam melhores resultados:

$$r(s_t, a_t) = \begin{cases} r_{alvo} \text{ if } d_t < c_d \\ r_{colide} \text{ if } min_x < c_o \\ r_{ocio} \text{ if } min_x >= c_o \text{ and } d_t >= c_d \end{cases} \tag{3.1}$$

Apenas três recompensas foram definidas, uma por realizar a tarefa corretamente e a segunda em caso de falha. O agente recebe 200 de recompensa r_{alvo} quando a meta é atingida em uma margem de c_d metros. Essa margem, expressamente, foi fixada em $0,25$ metros. Em um caso de colisão contra um obstáculo ou atingindo os limites do cenário, uma recompensa negativa r_{colide} de -20 é dada. As colisões são verificadas se as leituras do sensor de distância forem inferiores a uma distância c_o de $0,12$ metros. Enquanto isso, se o agente ficar longe do alvo por uma distância $d_t < c_d$ e seus resultados de laser forem expressos por min_x mantendo uma distância maior ou igual a c_d, uma recompensa r_{ocio} de 0 é dada. A mesma recompensa r_{ocio} é dada se a etapa do episódio terminar com as condições expressas pela equação 3.1. Esse sistema de recompensa simplificado também ajuda a focar nas abordagens Deep-RL, suas semelhanças e diferenças, em vez do cenário.

3.5 TURTLEBOT

O Turtlebot é um estilo de robô móvel montado para atender necessidades básicas de estudo de robótica. É muito utilizado nas universidades e seu kit de desenvolvimento faz parte do portfólio de produtos da linha de robótica educacional da empresa Robotis. Por ser um robô com baixo custo de hardware e software de código aberto, ele facilita a implementação de projetos, para uso em hobby, educação, prototipagem e pesquisa. Seus pacotes são fornecidos pela Robotis em seu site e-manual.

Existem diferentes modelos e versões de Turtlebot. Nesse experimento, foi escolhida a sua terceira versão, o TurtleBot3, no modelo Burguer, ilustrado na Figura 15 . O robô possui 2 motores DYNAMIXEL, um sensor laser LiDAR com 360 graus, um sensor IMU que mede a orientação do robô. O controle de baixo nível é feito com uma placa *OpenCR1.0* e um microprocessador Raspberry Pi 3.

As únicas alterações aplicadas aos pacotes estão no sensor LiDAR, onde a saída é reduzida para 24 espalhadas por 360 graus e essa modificação foi feita para reduzir o número de entradas nas redes.

Figura 15 – Turtlebot3

Fonte: https://www.roscomponents.com/en/mobile-robots/214-turtlebot-3.html

3.6 METODOLOGIA DE AVALIAÇÃO DE DESEMPENHO

O treinamento das redes DQN e Double DQN foram realizadas com o mesmo número de episódios em um mesmo ambiente para assegurar uma comparação mais coerente de resultados. Os treinamentos no Ambiente 1 foram realizados em 3000 episódios e, os treinamentos nos ambientes 2 e 3, em 5000 episódios. O primeiro ambiente precisa de menos episódios porque não possui obstáculos e isso faz com que a curva de aprendizagem dos modelos convirja mais rápido.

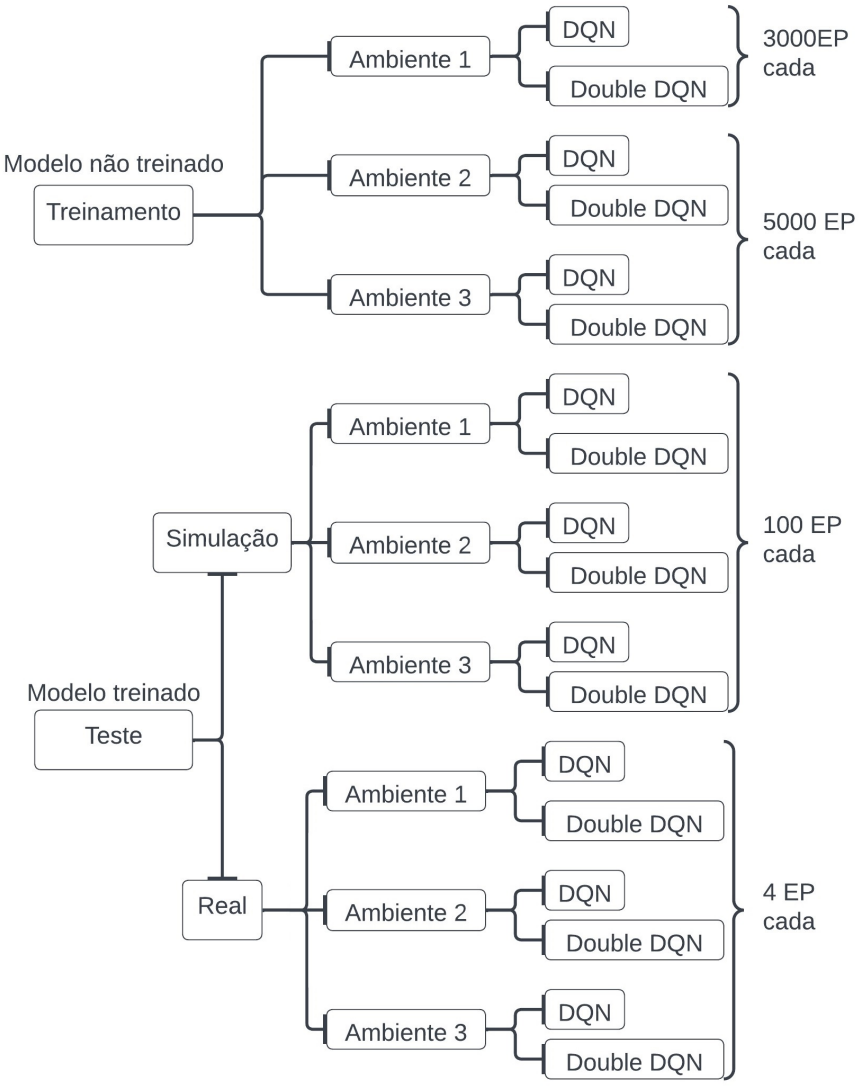
Durante o treinamento serão medidas as recompensas obtidas em cada episódio. Durante os testes será contabilizado o total de episódios que tiveram êxito em atingir o alvo, e contabilizar o tempo de duração de cada episódio para comparar as médias e desvio padrão de cada cenário.

4 APRESENTAÇÃO E DISCUSSÃO DOS RESULTADOS

Neste capítulo serão descritos os resultados obtidos projeto. A avaliação foi feita de duas formas, em simulação e em ambiente real.

Foram realizadas simulações com o Turtlebot 3 no Ambiente 1, 2 e 3, aplicando a rede DQN e Double DQN para treinamento do modelo, e foram registradas as recompensas obtidas. Após, realizou-se o teste em simulação do modelo nas mesmas condições e, por último, foi realizado o teste dos modelos nos ambientes reais, construídos em laboratório com as mesmas proposições dos ambientes simulado. O diagrama na Figura 16 ilustra a quantidade de episódios em cada etapa de treinamento e teste.

Figura 16 – Fluxograma das etapas de treinamento e teste simulado e real



Fonte: Autor

Foram realizados apenas 4 episódios para cada algoritmo em cada ambiente dos testes reais devido a limitação da aplicação desse teste. Como foi utilizada a visão computacional, há necessidade da comunicação da Jetson/câmera com o computador, e essa comunicação possui atraso. O processamento da imagem também acrescenta no tempo de atraso. Quando o atraso é maior que 3 segundos, os comandos que o robô recebe para alterar a velocidade angular já ficam obsoletos e a movimentação do robô não condiz com sua atual localização em relação ao alvo. Essa incoerência acontece porque a visão é utilizada justamente para, de forma simplificada, informar o robô sobre a distância e o ângulo que ele está em relação ao alvo. Apesar dos fatores apontados, foi possível capturar os dados de uma tentativa/episódio para cada alvo, somando os 4 episódios, já que são 4 alvos fixos.

4.1 SIMULAÇÃO E TREINAMENTO

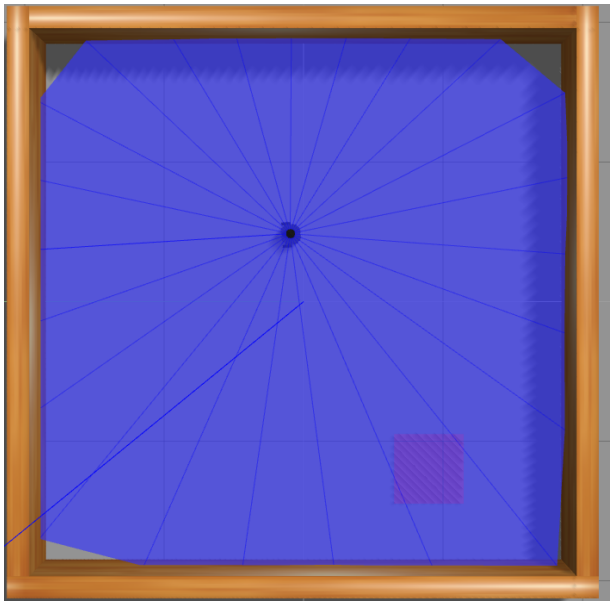
A primeira etapa realizada foi a simulação do Turtlebot 3 nos ambientes para treinamento dos modelos. Cada ambiente foi simulado duas vezes, uma vez para treinamento da rede DQN e outra para rede Double DQN.

O primeiro ambiente simulado em que o robô foi inserido é o ambiente 1, sem obstáculos. Na Figura 17 pode-se observar o Robô simulado em treinamento, utilizando seu sensor laser padrão. O objetivo em cada episódio desse treinamento é atingir a posição-objetivo, definida graficamente pelo quadrado vermelho no canto inferior direito. A áurea roxa é a visualização do alcance dos raios do laser.

A figura 18 apresenta o gráfico das recompensas obtidas na simulação de treinamento do Turtlebot 3 dentro do Ambiente 1 durante 3000 episódios. No gráfico, o eixo Y representa o valor das recompensas e o eixo X, os episódios. A linha azul mostra as recompensas no algoritmo DQN e a linha rosa no Double-DQN. A sombra transparente de colorida no gráfico representa o desvio padrão de cada curva.

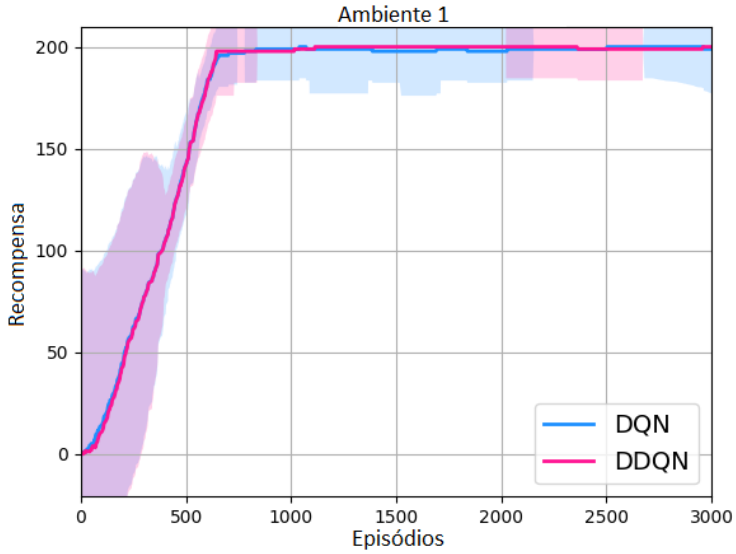
Pode-se observar que o valor das recompensas estabiliza próximo ao seu valor máximo de 200 por volta dos 600 episódios.

Figura 17 – Simulação do Turtlebot 3 no gazebo



Fonte: Autor

Figura 18 – Média Móvel das recompensas obtidas pelo Turtlebot3 em cada episódio de iteração no Ambiente 1.

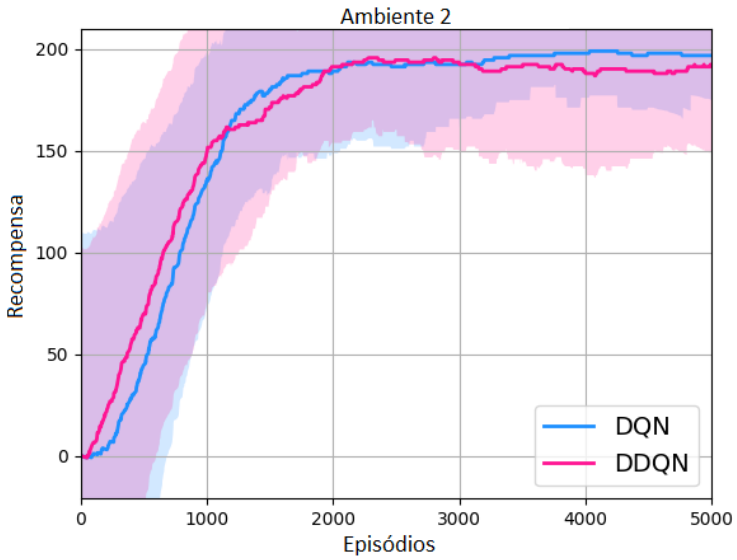


Fonte: Autor

A Figura 19 apresenta o gráfico de comparação das recompensas obtidas no treinamento do robô móvel no ambiente 2, entre os algoritmos DQN e Double DQN. Pode-se observar que a média de recompensar sobe mais rápido no treinamento com rede DDQN. O treinamento durou 5000 episódios e pode-se observar que o gráfico das recompensas quase estabiliza a partir dos 2000 episódios. Contudo, observa-se também uma maior variação do desvio padrão, o que indica que há mais recompensas variadas (de 0 ou -20),

indicando colisão ou esgotamento de tempo de navegação sem colidir ou atingir o alvo. Esse comportamento é observado devido aos obstáculos presentes no percurso.

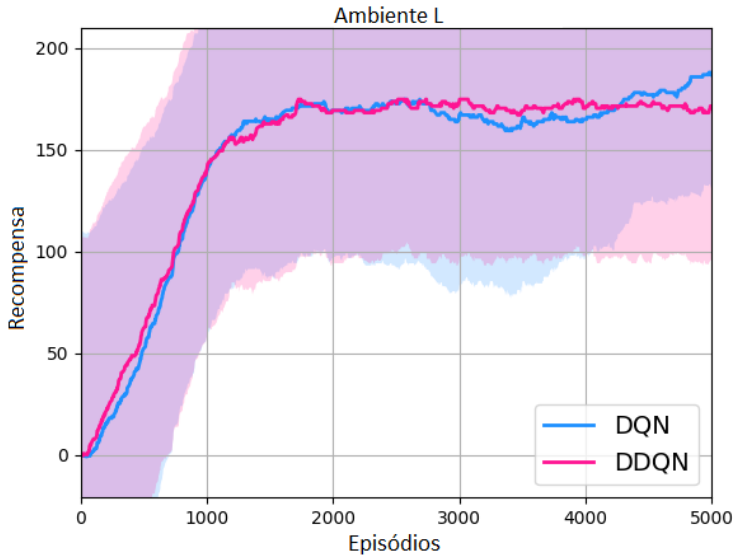
Figura 19 – Média Móvel das pontuações obtidas pelo Turtlebot3 em cada episódio no Ambiente 2.



Fonte: Autor

Após, foram realizadas as mesmas simulações mas com o Turtlebot 3 inserido no ambiente 3, com obstáculos retangulares. O resultado do desempenho do robô medido por sua pontuação ao longo de 5000 episódios é mostrado na Figura 20.

Figura 20 – Média Móvel das pontuações obtidas pelo Turtlebot3 em cada episódio no Ambiente 3.



Fonte: Autor

Pode-se observar que o gráfico de recompensas se mantém próximo a 175, indicando que o robô ainda apresenta colisões no treinamento mesmo após 5000 episódios. Ainda assim, é uma média que indica um bom aprendizado, tendo em vista a complexidade do ambiente.

4.2 TESTE EM SIMULAÇÃO E REAL

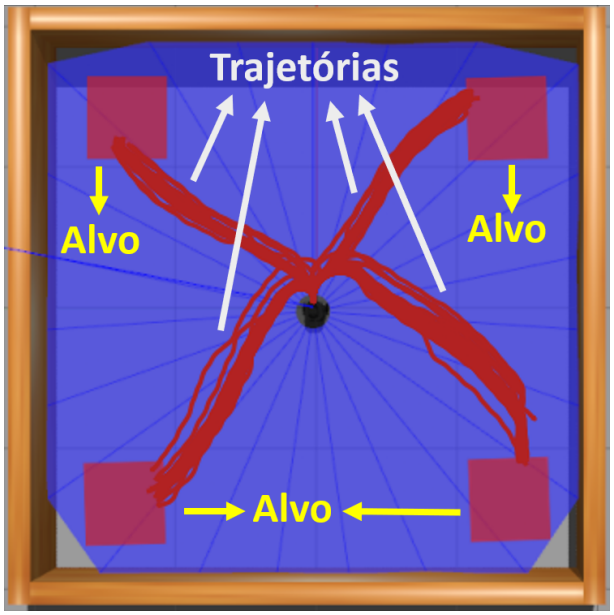
A etapa de testes foi realizada em simulação e em ambiente real. Os testes em ambiente simulado duram 100 episódios para cada cenário e algoritmo, e os testes reais duraram 4 episódios para cada condição.

Para os testes simulados, os modelos treinados foram carregados no Turtlebot3 simulado e o robô foi posto a navegar utilizando o modelo para tomar as decisões.

Foram fixados 4 alvos, um em cada canto do quadrado para os dois primeiros ambiente e 3 em cada canto mais um no meio dos obstáculos para o terceiro ambiente. Os alvos foram fixados para prevenir que algum trajeto seja beneficiado ou prejudicado pela possível localização de alvos aleatórios. Como cada teste foi realizado em 100 episódios, cada alvo esteve presente em 25 episódios.

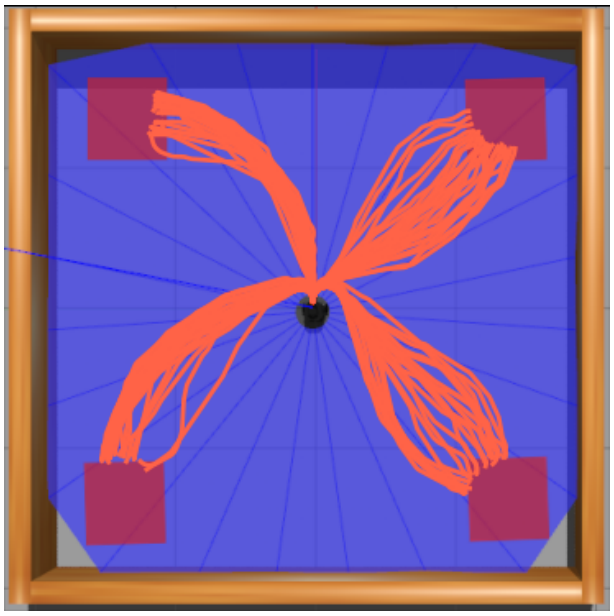
A Figura 21 mostra os resultados do teste no ambiente 1 com a rede DQN, e, a Figura 22, com a rede Double DQN. Os traços coloridos marcados na figura representam o caminho que o robô percorreu em cada episódio do teste.

Figura 21 – Trajetórias realizadas pelo Turtlebot3 no teste em simulação no Ambiente 1 com DQN.



Fonte: Autor

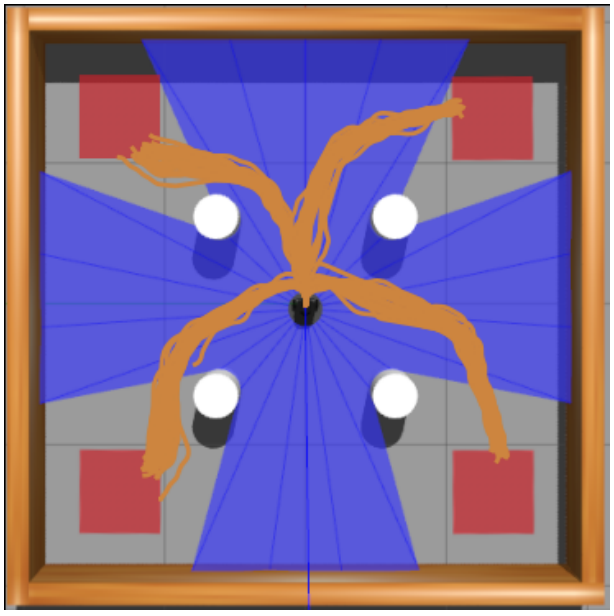
Figura 22 – Trajetórias realizadas pelo Turtlebot3 no teste em simulação no Ambiente 1 com DDQN.



Fonte: Autor

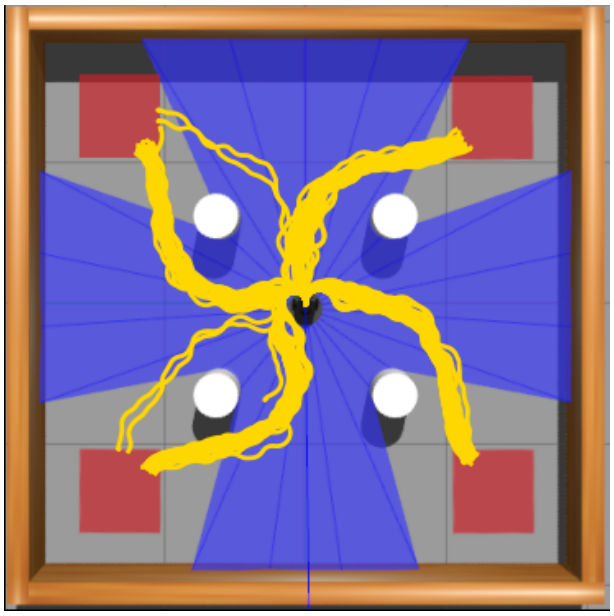
A Figura 23 mostra os trajetos de navegação do robô no ambiente 2, utilizando a rede DQN. Os percursos utilizando a rede Double DQN são mostrados na Figura 24.

Figura 23 – Trajetórias realizadas pelo Turtlebot3 no teste em simulação no Ambiente 2 com DQN.



Fonte: Autor

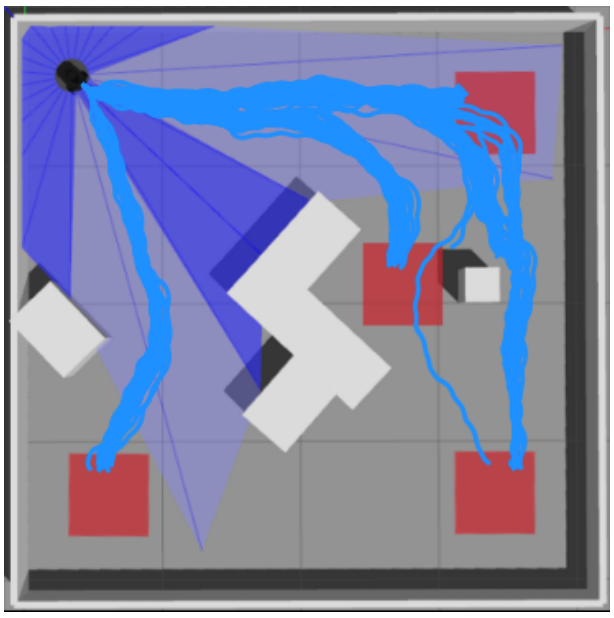
Figura 24 – Trajetórias realizadas pelo Turtlebot3 no teste em simulação no Ambiente 2 com DDQN.



Fonte: Autor

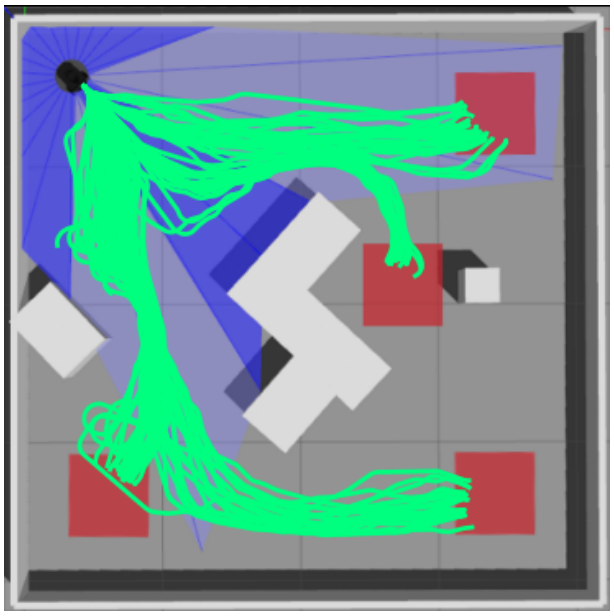
Nas Figuras 25 e 26 são apresentadas as trajetórias realizadas pelo robô nos testes em simulação, no Ambiente 3, dos modelos treinados do DQN e Double-DQN, respectivamente.

Figura 25 – Trajetórias realizadas pelo Turtlebot3 no teste em simulação no Ambiente 3 com DQN.



Fonte: Autor

Figura 26 – Trajetórias realizadas pelo Turtlebot3 no teste em simulação no Ambiente 3 com DDQN.



Fonte: Autor

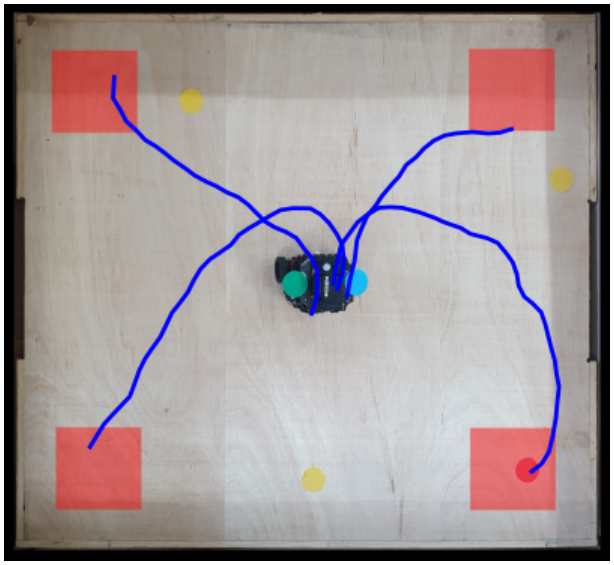
Analisando as trajetórias, é possível observar que o robô colide algumas vezes nos testes no Ambiente 3, que é mais complexo. Com o algoritmo DQN, o robô explorou menos o mapa e colidiu mais no pequeno quadrado brando na direita do ambiente. Já no DDQN, o Turtlebot explorou mais o ambiente, mas colidiu menos, sem colidir nenhuma vez no quadrado a direta que apresentou colisões no teste com o DQN.

4.3 TESTES REAIS

Foram realizados testes nos ambientes reais. Os alvos foram fixados nos mesmos lugares do teste simulado, para tornar a comparação mais confiável. Foi realizado um episódio para cada alvo em cada cenário e algoritmo, totalizando 4 tentativas para cada configuração.

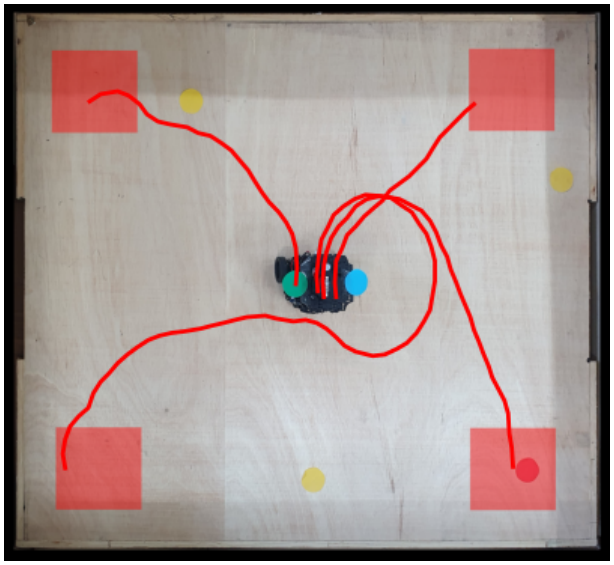
A Figura 27 mostra o resultado do teste em ambiente real, no ambiente 1, com a rede DQN. Na Figura 28, pode-se observar o trajeto realizado utilizando a rede Double DQN.

Figura 27 – Trajetórias realizadas pelo Turtlebot3 no teste no Ambiente 1 real com DQN.



Fonte: Autor

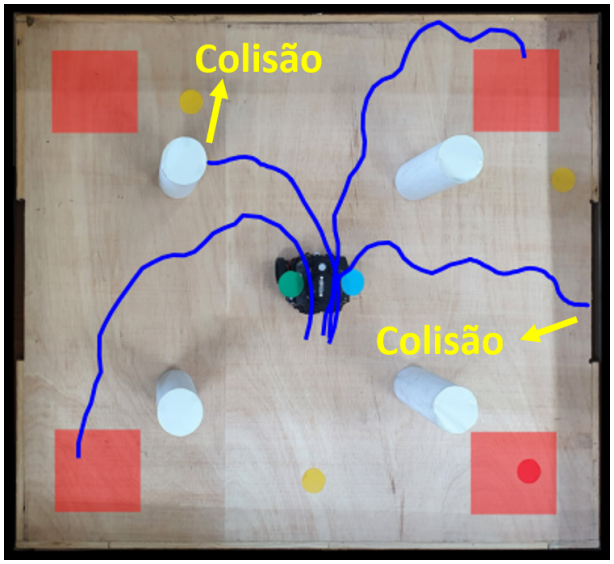
Figura 28 – Trajetórias realizadas pelo Turtlebot3 no teste no Ambiente 1 real com DDQN.



Fonte: Autor

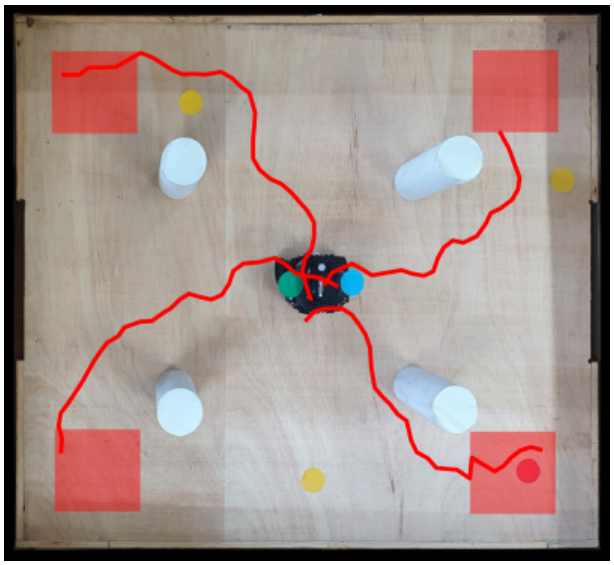
As Figuras 29 e 30 mostram a trajetória do robô no ambiente real 1, com algoritmo DQN e Double DQN, respectivamente.

Figura 29 – Trajetórias realizadas pelo Turtlebot3 no teste no Ambiente 2 real com DQN.



Fonte: Autor

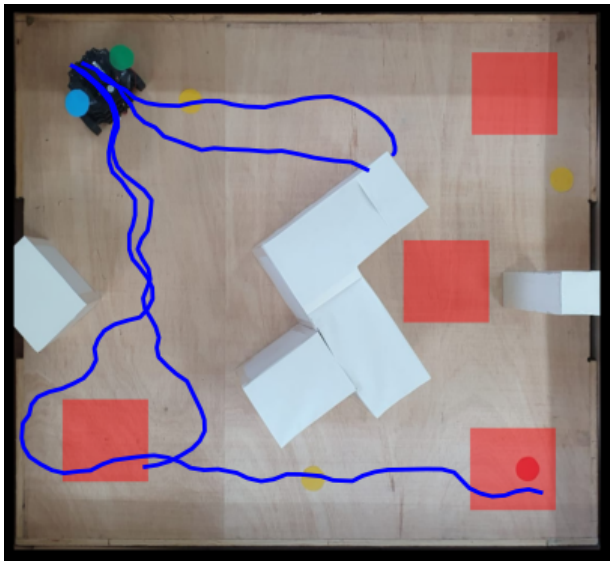
Figura 30 – Trajetórias realizadas pelo Turtlebot3 no teste no Ambiente 2 real com DDQN.



Fonte: Autor

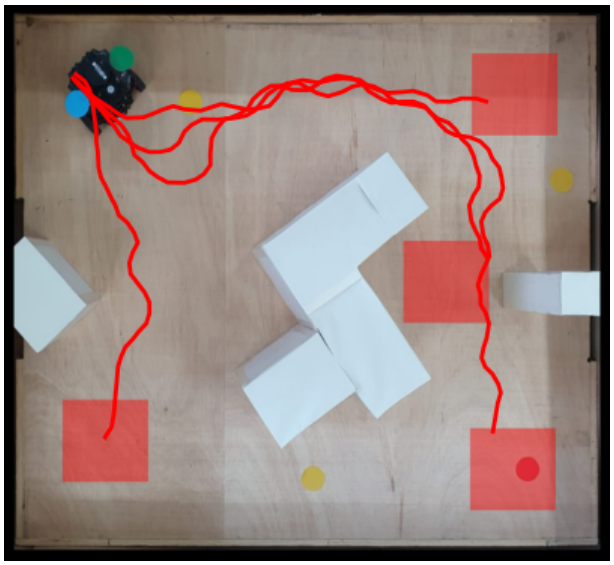
As trajetórias do robô no ambiente 3 real podem ser visualizadas nas Figuras 31 e 32, com a rede DQN e Double DQN, respectivamente.

Figura 31 – Trajetórias realizadas pelo Turtlebot3 no teste no Ambiente 3 real com DQN.



Fonte: Autor

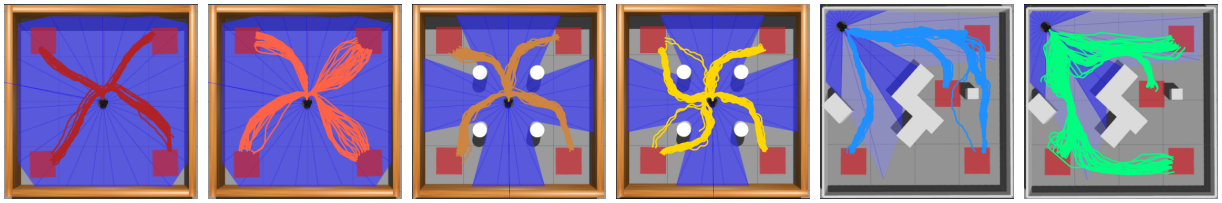
Figura 32 – Trajetórias realizadas pelo Turtlebot3 no teste no Ambiente 3 real com DDQN.



Fonte: Autor

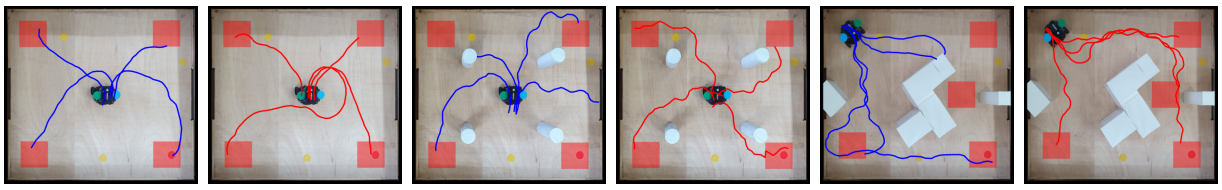
Analisando as trajetórias do robô, pode-se observar que o robô não seguiu os mesmos padrões dos testes simulados nos testes reais. Pode-se perceber também que o Turtlebot colidiu mais com o algoritmo DQN do que com o Double-DQN nos testes realizados.

Para a comparação visual das trajetórias, apresenta-se nas Figuras 33 e 34 o registro das trajetórias do robô em cada episódio de teste, simulado e real, respectivamente. Todas essas imagens já foram apresentadas anteriormente mas separadamente.



(a) DQN (b) DDQN (c) DQN (d) DDQN (e) DQN (f) DDQN

Figura 33 – Navegação de cada algoritmo em cada ambiente simulado em 100 tentativas.
Fonte: Autor



(a) DQN (b) DDQN (c) DQN (d) DDQN (e) DQN (f) DDQN

Figura 34 – Navegação de cada algoritmo em cada ambiente real em 4 tentativas.
Fonte: Autor

Para quantificar e analisar os resultados, foi registrado o número total de tentativas bem sucedidas em atingir o alvo. Também foram registradas as métricas de tempo médio de navegação e seus desvios padrão de 100 (para simulação) e 4 (para testes de navegação reais) episódios em três diferentes cenários para todas as abordagens.

Esses resultados gerais coletados referentes aos ambientes 1, 2 e 3, respectivamente, são mostrados nas Tabelas 2, 3 e 4. O Tempo do Episódio (TE) e a Taxa de Sucesso (TS) são apresentados para cada teste.

No ambiente sem obstáculos, os testes simulados e reais nos dois algoritmos tiveram taxa de 100% de sucesso, atingindo o alvo em todas as tentativas. Em simulação, o tempo médio de navegação com a rede Double DQN foi menos, enquanto no teste real o tempo médio com a mesma rede foi 1,61s mais lento. Contudo, observa-se que o desvio padrão nesse caso é de ± 6s e no algoritmo DQN é de ±2,61, podendo indicar uma inconsistência de resultados.

Tabela 2 – Comparação dos resultados dos testes reais e simulados.

Ambiente	**Algoritmo**	TE_{sim} **(s)**	TS_{sim}	TE_{real} **(s)**	TS_{real}
1	DQN	16.02 ± 1.67	100%	12.59 ± 2.61	100%
1	DDQN	15.72 ± 1.06	100%	14.20 ± 6.00	100%

Já no ambiente 2, com os obstáculos cilíndricos, a taxa de sucesso dos algoritmos em ambiente simulado seguem 100%, acertando o alvo nas 100 tentativas desse teste, mas em ambiente real a rede DQN performou 50%, acertando o alvo em duas das quatro

tentativas que abrangem esse teste. Essas métricas podem ser observadas na Tabela 3.

Tabela 3 – Comparação dos resultados dos testes reais e simulados.

Ambiente	**Algoritmo**	TE_{sim} **(s)**	TS_{sim}	TE_{real} **(s)**	TS_{real}
2	DQN	17.13 ± 0.28	100%	11.39 ± 4.24	50%
2	DDQN	17.39 ± 1.15	100%	20.11 ± 5.88	100%

No ambiente 3, que é um pouco mais complexo, a diferença entre os resultados dos testes dos modelos aparecem mais. Conforme mostrado na Tabela 4, os testes simulados obtiveram 89% de taxa de sucesso com o modelo DQN e 97% com o modelo Double DQN. No teste real, o DQN teve uma taxa de sucesso de 50%, atingindo o alvo em duas das quatro tentativas e do Double DQN de 100%, acertando o alvo em todas as tentativas do teste.

Tabela 4 – Comparação dos resultados dos testes reais e simulados.

Ambiente	**Algoritmo**	TE_{sim} **(s)**	TS_{sim}	TE_{real} **(s)**	TS_{real}
3	DQN	26.05 ± 6.94	89%	20.12 ± 9.38	50%
3	DDQN	25.60 ± 7.73	97%	22.28 ± 6.69	100%

Uma possível alternativa para aumentar o desempenho das redes é retirar a camada de Dropout da rede neural, já que essa função é utilizada para diminuir o *overfitting* da rede e nesse caso não é algo que é necessário evitar.

De maneira geral, pode-se observar que a rede Double DQN apresentou um melhor desempenho nos testes simulados e real. Nos ambientes 2 e 3, que apresentam obstáculos e portanto são mais complexos, essa diferença é maior. No ambiente 3, até mesmo os testes em simulação mostraram maior assertividadade em menos tempo com a rede Double DQN em comparação com a rede DQN.

5 CONCLUSÃO

Considerando os conhecimentos e simulações demonstradas no relatório, avaliasse que o trabalho está concluído. Foram aplicados e realizada a comparação do desempenho de dois algoritmos de aprendizagem pro reforço profundo na navegação de um robô móvel com desvio de trajetória, em ambiente real e de simulação.

Inseriu-se o Turtlebot 3 nos ambientes 1, 2 e 3 de simulação do Gazebo, sendo o ambiente 2 e 3 com obstáculos. Foi realizada a navegação nos ambientes em simulação e nos ambiente reais, construídos em laboratório.

Foi realizado o treinamento das redes Deep Q-Network e Double DQN em simulação nos 3 ambientes. Para visualização e avaliação da performance de aprendizagem, a média móvel das recompensas de cada episódio foi registrada em um gráfico para cada treinamento. Assim, observou-se que as redes obtiveram bom aprendizado nos três treinamentos.

Também testou-se as redes DQN e DDQN, já treinadas, nas simulações do Turtlebot 3 nos ambientes 1, 2 e 3, e, finalmente, realizou-se os testes nos três ambientes reais construídos em laboratório. Para comparar visualmente a performance da rede nos testes em simulação e reais, foram registradas as trajetórias de navegação do robô móvel em cada ambiente e algoritmo. Além disso, o tempo dos episódios e as taxas de sucesso em atingir o alvo também foram medidas.

Como sugestão para trabalhos futuros, pode-se aplicar a rede Dueling Deep Q-Network para comparação das performances em navegação de robôs móveis.

REFERÊNCIAS BIBLIOGRÁFICAS

ANGGRAENI, P.; ROKHIM, I.; SALAM, R. M. Design and development of multiple mobile manipulator robots using gazebo-ros. In: **2020 International Conference on Applied Science and Technology (iCAST)**. [S.l.: s.n.], 2020. p. 672–676.

CHEN, Y. F. et al. Socially aware motion planning with deep reinforcement learning. In: IEEE. **Intelligent Robots and Systems (IROS), 2017 IEEE/RSJ International Conference on**. [S.l.], 2017. p. 1343–1350.

DAI, Z. et al. Transformer-xl: Attentive language models beyond a fixed-length context. **CoRR**, abs/1901.02860, 2019.

DOBREVSKI, M.; SKOCAJ, D. Map-less goal-driven navigation based on reinforcement learning. In: **23rd Computer Vision Winter Workshop**. [S.l.: s.n.], 2018.

FAIRCHILD, C.; HARMAN, T. L. **ROS Robotics By Example - Second Edition: Learning to Control Wheeled, Limbed, and Flying Robots Using ROS Kinetic Kame**. 2nd. ed. [S.l.]: Packt Publishing, 2017. ISBN 1788479599.

FIGUEIREDO, J. M. P.; REJAILI, R. P. A. **Aplicação de algoritmos de aprendizagem por reforço para controle de navios em águas restritas**. 2018 — Curso de Graduação em Engenharia Mecânica de Automação e Sistemas, Universidade Federal de São Paulo, São Paulo, 2018.

GERON, A. **Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems**. 1st. ed. [S.l.]: O'Reilly Media, 2017. ISBN 978-1491962299.

_____. **Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems**. 2nd. ed. [S.l.]: O'Reilly Media, 2019. ISBN 9781492032649.

HASSELT, H. Double q-learning. **Advances in neural information processing systems**, v. 23, 2010.

HASSELT, H. V.; GUEZ, A.; SILVER, D. Deep reinforcement learning with double q-learning. In: **Thirtieth AAAI Conference on Artificial Intelligence**. [S.l.: s.n.], 2016.

JESUS, J. C. et al. Deep deterministic policy gradient for navigation of mobile robots in simulated environments. In: IEEE. **2019 19th International Conference on Advanced Robotics (ICAR)**. [S.l.], 2019. p. 362–367.

LASKIN, M.; SRINIVAS, A.; ABBEEL, P. Curl: Contrastive unsupervised representations for reinforcement learning. In: PMLR. **International Conference on Machine Learning**. [S.l.], 2020. p. 5639–5650.

LILLICRAP, T. P. et al. Continuous control with deep reinforcement learning. **arXiv preprint arXiv:1509.02971**, 2015.

MNIH, V. et al. Playing atari with deep reinforcement learning. **arXiv preprint arXiv:1312.5602**, 2013.

_____. Human-level control through deep reinforcement learning. **nature**, Nature Publishing Group, v. 518, n. 7540, p. 529–533, 2015.

PFITSCHER, M. et al. Article users activity gesture recognition on kinect sensor using convolutional neural networks and fastdtw for controlling movements of a mobile robot. **Inteligencia Artificial**, v. 22, n. 63, p. 121–134, Apr. 2019.

RAO, D.; MCMAHAN, B. **Natural language processing with PyTorch: build intelligent language applications using deep learning**. [S.l.]: "O'Reilly Media, Inc.", 2019.

SILVA, R. M. da; CUADROS, M. A. de S. L.; GAMARRA, D. F. T. Comparison of a backstepping and a fuzzy controller for tracking a trajectory with a mobile robot. In: ABRAHAM, A. et al. (Ed.). **Intelligent Systems Design and Applications**. Cham: Springer International Publishing, 2020. p. 212–221.

Stanford Artificial Intelligence Laboratory et al. **Robotic Operating System**. Disponível em: <https://www.ros.org>.

SUBRAMANIAN, V. **Deep Learning with PyTorch: A practical approach to building neural network models using PyTorch**. [S.l.]: Packt Publishing Ltd, 2018.

SUTTON, R. S.; BARTO, A. G. **Reinforcement learning: An introduction**. [S.l.]: MIT press, 2018.

TAI, L.; LIU, M. A robot exploration strategy based on q-learning network. In: IEEE. **2016 ieee international conference on real-time computing and robotics (rcar)**. [S.l.], 2016. p. 57–62.

_____. Towards cognitive exploration through deep reinforcement learning for mobile robots. **CoRR**, abs/1610.01733, 2016.

TAI, L.; PAOLO, G.; LIU, M. Virtual-to-real deep reinforcement learning: Continuous control of mobile robots for mapless navigation. In: IEEE. **Intelligent Robots and Systems (IROS), 2017 IEEE/RSJ International Conference on**. [S.l.], 2017. p. 31–36.

WU, C.-J. et al. Machine learning at facebook: Understanding inference at the edge. In: IEEE. **2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)**. [S.l.], 2019. p. 331–344.

XU, Z.-x. et al. Deep reinforcement learning with sarsa and q-learning: a hybrid approach. **IEICE TRANSACTIONS on Information and Systems**, The Institute of Electronics, Information and Communication Engineers, v. 101, n. 9, p. 2315–2322, 2018.

ANEXO A – CÓDIGO DQN

```
#!/usr/bin/env python3

import rospy
import os
import numpy as np
import time
import sys
import csv

from tqdm import tqdm
sys.path.append(os.path.dirname(os.path.abspath(os.path.dirname(__file__))))
from std_msgs.msg import Float32MultiArray

import gym
import math
import gym_turtlebot3
from geometry_msgs.msg import Twist, Point, Pose
from sensor_msgs.msg import LaserScan
from nav_msgs.msg import Odometry
from std_srvs.srv import Empty
from gym import spaces
from gym.utils import seeding
from std_msgs.msg import Float32MultiArray

EPISODES = 5001

import torch as T

import numpy as np

import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.tensorboard import SummaryWriter

time.sleep(4)
```

```
os.environ['ROS_MASTER_URI'] = "http://localhost:{}/".format(11310 + 1)
rospy.init_node('TurtleBot3_Circuit_Simple-v0'.replace('-', '_') +
                "_w{}".format(1))
env = gym.make('TurtleBot3_Circuit_Simple-v0', observation_mode=0,
                continuous=True, env_stage=1)
time.sleep(4)

observation = env.reset(new_random_goals=True, goal=None)

if not os.path.exists('logs'):
os.makedirs('logs')

writer = SummaryWriter('logs')


class ReplayBuffer():
    def __init__(self, max_size, input_dims, batch_size):
        self.mem_size = max_size
        self.mem_cntr = 0
        self.batch_size = batch_size

        self.state_memory  = np.zeros((self.mem_size, *input_dims),
                                      dtype = np.float32)
        self.new_state_memory = np.zeros ((self.mem_size, *input_dims),
                                          dtype =np.float32)
        self.action_memory = np.zeros (self.mem_size, dtype =np.int32)

        self.reward_memory = np.zeros (self.mem_size, dtype =np.float32)
        self.terminal_memory = np.zeros (self.mem_size, dtype =bool)

    def store_transition (self, state, action, reward, new_state , done):

        index = self.mem_cntr \% self.mem_size

        self.state_memory[index]= state
        self.new_state_memory[index] = new_state
        self.action_memory[index] = action
        self.reward_memory [index] = reward
        self.terminal_memory[index] = done
```

```
        self.mem_cntr+=1

    def sample_buffer (self, batch_size):

        max_mem = min (self.mem_cntr, self.mem_size)

        batch = np.random.choice(max_mem, batch_size, replace = False)

        batch_index = np.arange (self.batch_size, dtype=np.int32)

        states = self.state_memory[batch]
        states_=  self.new_state_memory[batch]
        rewards =  self.reward_memory[batch]
        actions =  self.action_memory[batch]
        terminal = self.terminal_memory[batch]

        return states, actions, rewards, states_, terminal, batch_index

class LinearDeepQNetwork(nn.Module):

    def __init__(self, lr, n_actions, input_dims):

        super(LinearDeepQNetwork,self).__init__()

        self.fc1 = nn.Linear(*input_dims, 256)
        self.fc2 = nn.Linear (256, 256)
        self.dropout = nn.Dropout(0.2)
        self.fc3 = nn.Linear (256, n_actions)

        self.optimizer = optim.Adam(self.parameters(),lr = lr)
        self.loss = nn.MSELoss()

        self.device = T.device('cuda:0' if T.cuda.is_available()
                               else 'cpu')

        self.to(self.device)

    def forward(self, state):
```

```
        layer1 = F.relu (self.fc1(state))
        layer2 = F.relu(self.fc2(layer1))
        drop_out = self.dropout(layer2)

        out_actions = self.fc3(drop_out)

        return out_actions

class ReinforceAgent():
    def __init__(self, state_size, action_size, writer):
        self.pub_result = rospy.Publisher('result', Float32MultiArray,
                                          queue_size=5)
        self.Path = os.path.dirname(os.path.realpath(__file__))
        self.dirPath = self.Path.replace('turtlebot3_dqn/nodes',
                       'turtlebot3_dqn/save_model/torch_model/stage_4_')
        self.resultPATH = self.Path.replace('turtlebot3_dqn/nodes',
                        'turtlebot3_dqn/result/result.csv')
        self.result = Float32MultiArray()

        self.load_model = False
        self.load_episode = 0
        self.state_size = state_size
        self.observation_space = (self.state_size,)

        self.action_size = action_size
        self.action_space = np.arange(0,5,1)

        self.day = 1905

        self.writer = writer

        '''Hyperparameters'''

        self.episode_step = 6000
        self.target_update = 2000
        self.discount_factor = 0.99
        self.learning_rate = 0.00025
        self.epsilon = 1.0
        self.epsilon_decay = 0.99
```

```
        self.epsilon_min = 0.05
        self.batch_size = 64
        self.train_start = self.batch_size

        self.global_step = 0

        self.mem_size = 1000000
        self.memory = ReplayBuffer(self.mem_size,
                                   self.observation_space,
                                   self.batch_size)

        self.model = LinearDeepQNetwork(self.learning_rate,
                                        self.action_size,
                                        self.observation_space)
        self.target_model = LinearDeepQNetwork(self.learning_rate,
                                               self.action_size,
                                               self.observation_space)

        print(self.model)

        self.updateTargetModel()

        if self.load_model:
            self.epsilon = 0.3
            self.global_step = self.load_episode
            print ('Loading model at episode: ', self.load_episode)
            self.model = T.load(self.dirPath +
                                str(self.load_episode) + '.pt')
            self.model.eval()
            print ("Load model state dict: ", self.model.state_dict())

    def updateTargetModel(self):
        self.target_model.load_state_dict(self.model.state_dict())
        print ('Updated Target Model')

    def choose_action (self, observation):
        if np.random.random() < self.epsilon:
            action = np.random.choice(self.action_space)
        else:
```

```
        state = T.tensor(observation,
                         dtype= T.float).to(self.model.device)
        actions = self.model(state)

        action =T.argmax(actions).item()
    return action

def store_trasition(self, state, action, reward, new_state, done):
    self.memory.store_transition(state, action, reward,
                                 new_state, done)

def learn(self):
    if (self.memory.mem_cntr < self.train_start):
        print ("------Not training---------")
        return

    self.model.optimizer.zero_grad()

    if self.global_step <= agent.target_update:
        target = False
    else:
        target = True



    states, actions, rewards, states_ ,terminals, batch_index =
              self.memory.sample_buffer(self.memory.batch_size)

    states = T.tensor(states, dtype= T.float).to(self.model.device)
    states_ = T.tensor(states_, dtype= T.float).to(self.model.device)
    rewards = T.tensor(rewards).to(self.model.device)
    terminals = T.tensor(terminals).to(self.model.device)

    q_prediction = self.model(states)[batch_index, actions]

    if target:
        q_next = self.target_model(states_)
    else:
        q_next = self.model(states_)
```

```
        q_next [terminals] = 0.0

        q_target = rewards + self.discount_factor * T.max(q_next,dim =1) [0]

        loss = self.model.loss(q_target, q_prediction).to(self.model.device)
        loss.backward()
        self.model.optimizer.step()
        self.writer.add_scalar("Loss/train", loss, self.global_step)

        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay
        else:
            self.epsilon =  self.epsilon_min


if __name__ == '__main__':
    pub_result = rospy.Publisher('result', Float32MultiArray,
                                 queue_size=5)
    pub_get_action = rospy.Publisher('get_action', Float32MultiArray,
                                     queue_size=5)
    result = Float32MultiArray()
    get_action = Float32MultiArray()

    state_size = 26
    action_size = 5

    det = {0: -1.5, 1: -0.75, 2: 0, 3: 0.75, 4: 1.5}


    agent = ReinforceAgent(state_size, action_size, writer)
    scores, episodes = [], []
    agent.global_step = 0
    start_time = time.time()

    for e in tqdm(range(agent.load_episode + 1, EPISODES)):
        done = False
        state = env.reset(new_random_goals=True, goal=None)
        score = 0
```

```
for t in range(agent.episode_step):

    action = agent.choose_action(state)
    next_state, reward, done, info = env.step(np.array([det[action],
                                    0.15], dtype=np.float))
    agent.store_trasition(state, action, reward, next_state, done)
    score += reward
    agent.learn()
    state = next_state

    get_action.data = [action, score, reward]
    pub_get_action.publish(get_action)

    if t >= 500:
        print("Time out!!")
        done = True

    if done:
        if e \% 100 == 0:
            T.save(agent.model.state_dict(), 'dqn_st1_model.pth')
            print ('Saved model at episode', e)
        agent.updateTargetModel()
        scores.append(score)
        episodes.append(e)
        m, s = divmod(int(time.time() - start_time), 60)
        h, m = divmod(m, 60)

        writer.add_scalar("Reward/train", reward, e)

        param_keys = ['epsilon']
        param_values = [agent.epsilon]
        param_dictionary = dict(zip(param_keys, param_values))
        break

    agent.global_step += 1
    if agent.global_step \% agent.target_update == 0:
        rospy.loginfo("UPDATE TARGET NETWORK")
        agent.updateTargetModel()
```

ANEXO B – CÓDIGO DDQN

```
#!/usr/bin/env python3

import rospy
import os
import numpy as np
import time
import sys
import csv

from tqdm import tqdm
sys.path.append(os.path.dirname(os.path.abspath(os.path.dirname(__file__))))
from std_msgs.msg import Float32MultiArray

import gym
import math
import gym_turtlebot3
from geometry_msgs.msg import Twist, Point, Pose
from sensor_msgs.msg import LaserScan
from nav_msgs.msg import Odometry
from std_srvs.srv import Empty
from gym import spaces
from gym.utils import seeding
from std_msgs.msg import Float32MultiArray

EPISODES = 5001

import torch as T

import numpy as np

import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.tensorboard import SummaryWriter

time.sleep(4)
```

```
os.environ['ROS_MASTER_URI'] = "http://localhost:{}/".format(11310 + 1)
rospy.init_node('TurtleBot3_Circuit_Simple-v0'.replace('-', '_') +
                "_w{}".format(1))
env = gym.make('TurtleBot3_Circuit_Simple-v0', observation_mode=0,
                continuous=True, env_stage=1)
time.sleep(4)

observation = env.reset(new_random_goals=True, goal=None)

if not os.path.exists('logs'):
os.makedirs('logs')

writer = SummaryWriter('logs')


class ReplayBuffer():
    def __init__(self, max_size, input_dims, batch_size):
        self.mem_size = max_size
        self.mem_cntr = 0
        self.batch_size = batch_size

        self.state_memory  = np.zeros((self.mem_size, *input_dims),
                                      dtype = np.float32)
        self.new_state_memory = np.zeros ((self.mem_size, *input_dims),
                                          dtype =np.float32)
        self.action_memory = np.zeros (self.mem_size, dtype =np.int32)

        self.reward_memory = np.zeros (self.mem_size, dtype =np.float32)
        self.terminal_memory = np.zeros (self.mem_size, dtype =bool)

    def store_transition (self, state, action, reward, new_state , done):

        index = self.mem_cntr \% self.mem_size

        self.state_memory[index]= state
        self.new_state_memory[index] = new_state
        self.action_memory[index] = action
        self.reward_memory [index] = reward
        self.terminal_memory[index] = done
```

```
        self.mem_cntr+=1

    def sample_buffer (self, batch_size):

        max_mem = min (self.mem_cntr, self.mem_size)

        batch = np.random.choice(max_mem, batch_size, replace = False)

        batch_index = np.arange (self.batch_size, dtype=np.int32)

        states = self.state_memory[batch]
        states_=  self.new_state_memory[batch]
        rewards =  self.reward_memory[batch]
        actions =  self.action_memory[batch]
        terminal = self.terminal_memory[batch]

        return states, actions, rewards, states_, terminal, batch_index

class LinearDeepQNetwork(nn.Module):

    def __init__(self, lr, n_actions, input_dims):

        super(LinearDeepQNetwork,self).__init__()

        self.fc1 = nn.Linear(*input_dims, 256)
        self.fc2 = nn.Linear (256, 256)
        self.dropout = nn.Dropout(0.2)
        self.fc3 = nn.Linear (256, n_actions)

        self.optimizer = optim.Adam(self.parameters(),lr = lr)
        self.loss = nn.MSELoss()

        self.device = T.device('cuda:0' if T.cuda.is_available()
                               else 'cpu')

        self.to(self.device)

    def forward(self, state):
```

```
        layer1 = F.relu (self.fc1(state))
        layer2 = F.relu(self.fc2(layer1))
        drop_out = self.dropout(layer2)

        out_actions = self.fc3(drop_out)

        return out_actions

class ReinforceAgent():
    def __init__(self, state_size, action_size, writer):
        self.pub_result = rospy.Publisher('result', Float32MultiArray,
                                          queue_size=5)
        self.Path = os.path.dirname(os.path.realpath(__file__))
        self.dirPath = self.Path.replace('turtlebot3_dqn/nodes',
                       'turtlebot3_dqn/save_model/torch_model/stage_4_')
        self.resultPATH = self.Path.replace('turtlebot3_dqn/nodes',
                        'turtlebot3_dqn/result/result.csv')
        self.result = Float32MultiArray()

        self.load_model = False
        self.load_episode = 0
        self.state_size = state_size
        self.observation_space = (self.state_size,)

        self.action_size = action_size
        self.action_space = np.arange(0,5,1)

        self.day = 1905

        self.writer = writer

        '''Hyperparameters'''

        self.episode_step = 6000
        self.target_update = 2000
        self.discount_factor = 0.99
        self.learning_rate = 0.00025
        self.epsilon = 1.0
        self.epsilon_decay = 0.99
```

```
        self.epsilon_min = 0.05
        self.batch_size = 64
        self.train_start = self.batch_size

        self.global_step = 0

        self.mem_size = 1000000
        self.memory = ReplayBuffer(self.mem_size,
                                   self.observation_space,
                                   self.batch_size)

        self.model = LinearDeepQNetwork(self.learning_rate,
                                        self.action_size,
                                        self.observation_space)
        self.target_model = LinearDeepQNetwork(self.learning_rate,
                                               self.action_size,
                                               self.observation_space)

        print(self.model)

        self.updateTargetModel()

        if self.load_model:
            self.epsilon = 0.3
            self.global_step = self.load_episode
            print ('Loading model at episode: ', self.load_episode)
            self.model = T.load(self.dirPath +
                                str(self.load_episode) + '.pt')
            self.model.eval()
            print ("Load model state dict: ", self.model.state_dict())

    def updateTargetModel(self):
        self.target_model.load_state_dict(self.model.state_dict())
        print ('Updated Target Model')

    def choose_action (self, observation):
        if np.random.random() < self.epsilon:
            action = np.random.choice(self.action_space)
        else:
```

```
            state = T.tensor(observation,
                             dtype= T.float).to(self.model.device)
            actions = self.model(state)

            action =T.argmax(actions).item()
        return action

    def store_trasition(self, state, action, reward, new_state, done):
        self.memory.store_transition(state, action, reward,
                                     new_state, done)

    def learn(self):
        if (self.memory.mem_cntr < self.train_start):
            print ("------Not training---------")
            return

        self.model.optimizer.zero_grad()

        if self.global_step <= agent.target_update:
            target = False
        else:
            target = True

        states, actions, rewards, states_ ,terminals, batch_index =
self.memory.sample_buffer(self.memory.batch_size)

        states = T.tensor(states, dtype= T.float).to(self.model.device)
        states_ = T.tensor(states_, dtype= T.float).to(self.model.device)
        rewards = T.tensor(rewards).to(self.model.device)
        terminals = T.tensor(terminals,
                             dtype = T.float).to(self.model.device)

        q_prediction = self.model(states)
        actions = T.tensor(actions,
                           dtype= T.int64).to(self.model.device)

        q_s_a = q_prediction.gather(1, actions.unsqueeze(1)).squeeze()
        q_tp1_values = self.model(states_).detach()
        _, a_prime = q_tp1_values.max(1)
```

```
        q_target_tp1_values = self.target_model(states_).detach()

        q_target_s_a_prime = q_target_tp1_values.
                             gather(1, a_prime.unsqueeze(1))

        q_target_s_a_prime = q_target_s_a_prime.squeeze()

        q_target_s_a_prime = (1 - terminals) * q_target_s_a_prime

        q_target = rewards + self.discount_factor * q_target_s_a_prime

        loss = self.model.loss(q_s_a, q_target)

        loss.backward()
        self.model.optimizer.step()
        self.writer.add_scalar("Loss/train", loss, self.global_step)

        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay
        else:
            self.epsilon =  self.epsilon_min


if __name__ == '__main__':
    pub_result = rospy.Publisher('result', Float32MultiArray,
                                 queue_size=5)
    pub_get_action = rospy.Publisher('get_action', Float32MultiArray,
                                     queue_size=5)
    result = Float32MultiArray()
    get_action = Float32MultiArray()

    state_size = 26
    action_size = 5

    det = {0: -1.5, 1: -0.75, 2: 0, 3: 0.75, 4: 1.5}


    agent = ReinforceAgent(state_size, action_size, writer)
```

```
scores, episodes = [], []
agent.global_step = 0
start_time = time.time()

for e in tqdm(range(agent.load_episode + 1, EPISODES)):
    done = False
    state = env.reset(new_random_goals=True, goal=None)
    score = 0

    for t in range(agent.episode_step):

        action = agent.choose_action(state)
        next_state, reward, done, info =
            env.step(np.array([det[action],0.15], dtype=np.float))
        agent.store_trasition(state, action, reward, next_state, done)
        score += reward
        agent.learn()
        state = next_state

        get_action.data = [action, score, reward]
        pub_get_action.publish(get_action)

        if t >= 500:
            print("Time out!!")
            done = True

        if done:
            if e \% 100 == 0:
                T.save(agent.model.state_dict(), 'dqn_st1_model.pth')
                print ('Saved model at episode', e)
            agent.updateTargetModel()
            scores.append(score)
            episodes.append(e)
            m, s = divmod(int(time.time() - start_time), 60)
            h, m = divmod(m, 60)

            writer.add_scalar("Reward/train", reward, e)

            param_keys = ['epsilon']
```

```
        param_values = [agent.epsilon]
        param_dictionary = dict(zip(param_keys, param_values))
        break

    agent.global_step += 1
    if agent.global_step \% agent.target_update == 0:
        rospy.loginfo("UPDATE TARGET NETWORK")
        agent.updateTargetModel()
```

NUP: 23081.108675/2022-62 **Prioridade:** Normal

Homologação de ata de defesa de TCC e estágio de graduação

125.322 - Bancas examinadoras de TCC: indicação e atuação

COMPONENTE

Ordem	Descrição	Nome do arquivo
13	Trabalho de Conclusão de Curso Versão final	TCC_Linda_Pos_Banca-1.pdf

Assinaturas

05/12/2022 10:39:56

DANIEL FERNANDO TELLO GAMARRA (PROFESSOR DO MAGISTÉRIO SUPERIOR)
07.54.00.00.0.0 - DEPARTAMENTO DE PROCESSAMENTO DE ENERGIA ELÉTRICA - DPEE

05/12/2022 12:09:03

LINDA DOTTO DE MORAES (Aluno de Graduação)
07.09.07.01.0.0 - Curso Engenharia de Controle e Automação - 121630