UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
CURSO DE GRADUAÇÃO EM ENGENHARIA ELETRÔNICA

Jonattan Alves de Carvalho

Modelagem e Controle de um Robô do Tipo Pêndulo Invertido Utilizando Aprendizado por Reforço

Florianópolis
2022

Jonattan Alves de Carvalho

Modelagem e Controle de um Robô do Tipo Pêndulo Invertido Utilizando Aprendizado por Reforço

Trabalho de Conclusão de Curso submetido ao Curso de Graduação em Engenharia Eletrônica da Universidade Federal de Santa Catarina para a obtenção do Título de Bacharel em Engenharia Eletrônica.
Orientador: Prof. Miguel Moreto, Dr.

Florianópolis
2022

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Carvalho, Jonattan Alves de
Modelagem e controle de um robô do tipo pêndulo invertido utilizando aprendizado por reforço / Jonattan Alves de Carvalho ; orientador, Miguel Moreto, 2022.
93 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Engenharia Eletrônica, Florianópolis, 2022.

Inclui referências.

1. Engenharia Eletrônica. 2. Equilíbrio de robôs. 3. Controle por aprendizado por reforço. 4. Modelagem de robô do tipo pêndulo invertido. I. Moreto, Miguel. II. Universidade Federal de Santa Catarina. Graduação em Engenharia Eletrônica. III. Título.

Jonattan Alves de Carvalho

Modelagem e Controle de um Robô do Tipo Pêndulo Invertido Utilizando Aprendizado por Reforço

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de Bacharel em Engenharia Eletrônica e aprovado em sua forma final pelo Curso de Graduação em Engenharia Eletrônica da Universidade Federal de Santa Catarina.

Florianópolis, 05 de agosto de 2022.

Documento assinado digitalmente
Fernando Rangel de Sousa
Data: 09/08/2022 21:26:35-0300
CPF: 884.649.114-91
Verifique as assinaturas em https://v.ufsc.br

Prof. Fernando Rangel de Sousa, Dr.
Coordenador do Curso de Graduação em Engenharia Eletrônica

Banca Examinadora:

Documento assinado digitalmente
Miguel Moreto
Data: 09/08/2022 18:20:50-0300
CPF: 948.850.100-63
Verifique as assinaturas em https://v.ufsc.br

Prof. Miguel Moreto, Dr.
Orientador
Universidade Federal de Santa Catarina

Documento assinado digitalmente
Danilo Silva
Data: 09/08/2022 19:49:32-0300
CPF: 031.557.924-26
Verifique as assinaturas em https://v.ufsc.br

Prof. Danilo Silva, Dr.
Universidade Federal de Santa Catarina

Documento assinado digitalmente
ANTONIO FELIPE DA CUNHA DE AQUINO
Data: 09/08/2022 17:28:55-0300
CPF: 044.652.117-59
Verifique as assinaturas em https://v.ufsc.br

Prof. Antonio Felipe da Cunha de Aquino, Dr.
Universidade Federal de Santa Catarina

RESUMO

Este trabalho trata de uma pesquisa experimental de simulação virtual com um modelo robótico do tipo pêndulo invertido sobre duas rodas paralelas. O equilíbrio deste sistema compõe uma tarefa clássica para a área de controle. Neste sistema, devido as propriedades como não linearidade e instabilidade, existem dificuldades para a aplicação de diversos algoritmos de controle clássico. Neste contexto, propõe-se uma abordagem alternativa para este sistema, com controladores baseados em inteligência artificial. O robô foi modelado em espaço de estados com base em leis de Newton e circuitos elétricos, incluindo parâmetros como momentos de inércia, indutância do motor, forças de atrito e efeitos da redução mecânica. Para tornar o modelo mais realista, os parâmetros do motor foram determinados a partir de um motor real. Neste modelo, o objetivo foi controlar tanto a inclinação como a posição do robô, definida em relação ao plano em que este se desloca. Para isso, aplicou-se dois algoritmos de aprendizado por reforço, *Double Deep Q-Network*, que utiliza redes neurais artificiais para tratar o espaço de estados de forma contínua e *Twin Delayed Deep Deterministic Policy Gradient*, que atua com estados e ações em domínio contínuo. Desta forma, solucionou-se a tarefa de controle através de aprendizado por tentativa e erro, sem necessidade de conhecimentos prévios sobre o sistema. A performance dos controladores resultantes, assim como a eficiência de seus treinamentos, foi avaliada com múltiplas simulações. Além disso, verificou-se o efeito de ruídos em medidas de sensores utilizados em aplicações práticas e a aplicação de um sinal de entrada para controlar a posição do robô. Por fim, avaliou-se efeitos de modificações na função que caracteriza o objetivo dos controladores. Assim, foi concluído que os algoritmos aplicados possuem potencial para superar controladores clássicos em diversos aspectos. Porém, estes métodos também apresentam dificuldades em suas implementações.

Palavras-chave: pêndulo invertido; modelo; controle; inteligência artificial; aprendizado de máquina; aprendizado por reforço.

ABSTRACT

This work presents an experimental research of virtual simulation with a robotic model of the type inverted pendulum on two parallel wheels. The balance of this system is a classic task for the control area. In this system, due to properties such as nonlinearity and instability, there are difficulties in the application of several classical control algorithms. In this context, an alternative approach to this system is proposed, with controllers based on artificial intelligence. The robot was modeled in state space based on Newton's laws and electrical circuits, including parameters such as moments of inertia, motor inductance, friction forces and effects of mechanical reduction. To make the model more realistic, the motor parameters were derived from a real engine. In this model, the objective was to control both the inclination and the position of the robot, defined in relation to the plane in which it moves. For this, two reinforcement learning algorithms were applied, Double Deep Q-Network, which uses artificial neural networks to treat the state space continuously and Twin Delayed Deep Deterministic Policy Gradient, which works with states and actions in a continuous domain. Therefore, the control task was solved through trial and error learning, without the need for previous knowledge about the system. The performance of the resulting controllers, as well as the efficiency of their training, was evaluated with multiple simulations. In addition, the effect of noise on sensor measurements used in practical applications and the application of an input signal to control the robot's position was verified. Finally, the effects of changes in the function that characterize the objective of the controllers were evaluated. Thus, it was concluded that the applied algorithms have the potential to overcome classical controllers in several aspects. However, these methods also present difficulties in their implementation.

Keywords: inverted pendulum; model; control; artificial intelligence; machine learning; reinforcement learning.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

BDF	*Backward Differentiation Formula*
CC	Corrente Contínua
DDPG	*Deep Deterministic Policy Gradient*
DQN	*Deep Q-Network*
EMF	*Electromotive Force*
IA	Inteligência Artificial
IMU	*Inertial Measurement Unit*
LQR	*Linear Quadratic Regulator*
LSODA	*Livermore Solver for Ordinary Differential Equations*
MDP	*Markov Decision Process*
ML	*Machine Learning*
MWP	*Mobile Wheeled Pendulum*
PD	*Proportional-Derivative*
PID	*Proportional-Integral-Derivative*
PWM	*Pulse Width Modulation*
RL	*Reinforcement Learning*
TD3	*Twin Delayed Deep Deterministic Policy Gradient*

LISTA DE SÍMBOLOS

Q	Estimativa de q
ε	Probabilidade de escolha de ação aleatória pela política exploratória
τ	Torque produzido pelo motor
i	Corrente no motor
K_t	Constante de torque do motor
e	Força eletromotriz do motor
φ	Posição angular do eixo do motor
K_e	Constante de velocidade do motor
v	Tensão no motor
J	Momento de inércia do eixo do motor em conjunto com as rodas do robô
b	Coeficiente de atrito dinâmico do motor
θ	Inclinação do pêndulo
l	Distância entre o centro de massa do pêndulo e o eixo das rodas do robô
r	Raio das rodas do robô
g	Aceleração da gravidade
m	Massa do pêndulo
H	Componente de força horizontal percebida pelo pêndulo
V	Componente de força vertical percebida pelo pêndulo
I	Momento de inércia do pêndulo
x	Posição do robô medida em relação ao plano em que este se desloca
$\mathbf{s}$	Vetor de estado
L	Indutância do motor
R	Resistência do motor
M	Massa de cada roda do robô
$\mathbf{A}$	Matriz jacobiana da equação de estado linearizada
r_t	Recompensa recebida no instante de tempo t
π	Política aprendida pelo agente de aprendizado por reforço
s_t	Estado do Processo de Decisão de Markov no instante de tempo t
a_t	Ação tomada pelo agente de aprendizado por reforço no instante de tempo t
S_t	Variável aleatória que representa o estado do Processo de Decisão de Markov no instante de tempo t
A_t	Variável aleatória que representa a ação executada no Processo de Decisão de Markov no instante de tempo t
R_t	Variável aleatória que representa a recompensa do estado do Processo de Decisão de Markov no instante de tempo t

λ_x	Parâmetro que define a importância do controle da varável x no processo de controle
λ_θ	Parâmetro que define a importância do controle da varável θ no processo de controle
G_t	Recompensa acumulada esperada
γ	Fator de desconto
q_π	Valor esperado de G_t ao seguir a política π
q	Valor esperado de G_t
q_*	Valor esperado de G_t ao seguir a política ótima
Q_π	Estimativa de q_π
α	Taxa de aprendizado
β	Política exploratória
N	Tamanho de *batch*
Q_e	Estimativa Q por redes neurais de avaliação
Q_t	Estimativa Q por redes neurais alvo
C	Intervalo de episódios para atualização de redes alvo
μ	Ação fornecida por redes neurais
J_π	Medida de performance da política π
Θ	Parâmetros de rede neural artificial
η	Ruído gaussiano
T	Fator de *Polyak Averaging*
D	Número de ações aleatórias aplicadas no método *Twin Delayed Deep Deterministic Policy Gradient*
d	Intervalo de episódios para atualização da rede atuadora do método *Twin Delayed Deep Deterministic Policy Gradient*
R_T	Recompensa do estado terminal
G_T	Recompensa acumulada associada ao estado terminal

SUMÁRIO

1 INTRODUÇÃO

O presente trabalho tem como proposta a elaboração de um controlador para robôs do tipo pêndulo invertido, também chamados de *mobile wheeled pendulum* (MWP) (BONARINI *et al.*, 2008), por meio de inteligência artificial (IA). Para isso, foi desenvolvido um modelo do sistema e aplicou-se dois métodos de aprendizado por reforço (RL, do inglês *Reinforcement Learning*) de maneira virtual.

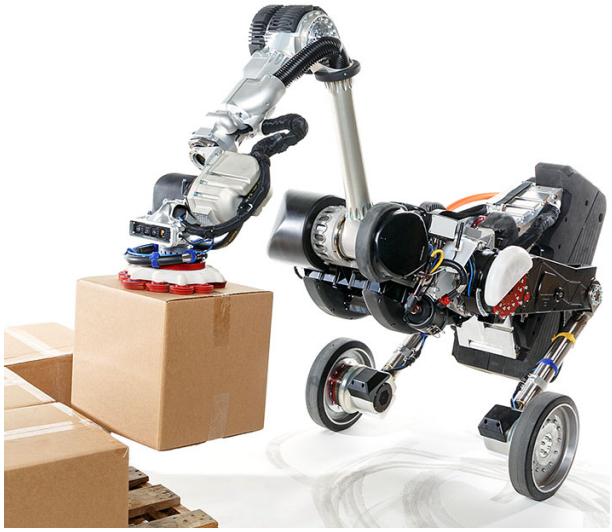
Um robô do tipo pêndulo invertido possui características de corpo suspenso de modo livre para balançar e seu centro de massa mantém-se acima do ponto de pivô (DURAND *et al.*, 2013). Estes são tipicamente compostos por um corpo acoplado a um carrinho ou, como neste trabalho, duas rodas.

Desta forma, neste modelo, deseja-se controlar tanto a inclinação do pêndulo invertido quanto a posição do robô. Esta posição é definida pela localização do centro das rodas do dispositivo em relação ao plano em que este se desloca.

1.1 MOTIVAÇÃO

Nos últimos anos, foram desenvolvidos diversos robôs com sistema de pêndulo invertido. Cita-se como exemplo Younis e Abdelati (2009) que projetaram um veículo com duas rodas paralelas para transporte humano individual e uso em ambientes externos e internos. A empresa Boston Dynamics, por sua vez, construiu um robô bípede com rodas, exibido na Figura 1, com o objetivo de transportar objetos em armazéns industriais para uma logística automatizada.

Figura 1 – Robô Handle da Boston Dynamics.



Fonte: Boston Dynamics[1].

[1] Disponível em: robots.ieee.org/robots/handle/?gallery=photo2. Acesso em: 20 jun. 2022.

Em ambos os casos citados, se comparados com robôs similares de quatro rodas, o sistema de equilíbrio em duas rodas contribui com a agilidade dos dispositivos e possibilita a locomoção em espaços reduzidos. Ademais, o pêndulo invertido pode modelar outras múltiplas aplicações, como lançamento de foguetes, edifícios sobre variação sísmica, posicionamento de satélites, embarcações e aeronaves (MÉNDEZ, 2015).

1.1.1 Sistema Pêndulo Invertido

O pêndulo invertido consiste em um sistema clássico, instável, não linear e muito utilizado para avaliação de técnicas de controle (BOUBAKER, 2013). Outra característica importante deste sistema compreende a sub-atuação, pois há duas saídas para se controlar a partir de apenas um sinal de controle (BOUBAKER, 2012). Os sinais de saída são a inclinação e a posição do dispositivo e, neste projeto, o sinal de controle corresponde à tensão aplicada no motores acoplados às rodas do robô.

Devido a estas características e às diversas aplicações deste sistema, o pêndulo invertido tem sido objeto de estudo para diversas pesquisas e está presente em múltiplos livros de controle, como os de Ogata (2010) e Khalil (1996). Para o estudo deste sistema, tipicamente se aplicam processos de linearização e simplificações. Porém, se considerados os efeitos não lineares, como forças de atrito e zonas mortas de movimento do motor, a tarefa de controle se torna mais complexa (REKDALSBAKKEN, 2006).

1.1.2 Inteligência Artificial

Com os recentes avanços computacionais, a área de IA e sua subárea aprendizado de máquina (ML, do inglês *Machine Learning*) têm crescido muito nos últimos anos (VERMA, A.; LAMSAL; VERMA, P., 2021). Em especial, *reinforcement learning* constitui um tópico de ML que tem sido muito utilizado para resolver problemas de controle, inclusive de pêndulo invertido.

Enquanto métodos como aprendizado supervisionado, subárea de *machine learning*, atuam a partir de dados disponibilizados pelo desenvolvedor, agentes de RL são treinados ao mesmo tempo que adquirem dados de um ambiente (KOBER; BAGNELL; PETERS, 2013).

Uma habilidade importante de RL abrange a capacidade de aprender a resolver um problema sem conhecimento prévio do sistema (KOBER; BAGNELL; PETERS, 2013). Devido a esta abordagem, diversas características do pêndulo invertido, que dificultam a aplicação de métodos clássicos de controle, são menos significativas para o desenvolvimento ou performance de algoritmos de RL (VICHUGOV; TSAPKO, G. P.; TSAPKO, S. G., 2005).

Deste modo, o uso de RL foi motivado tanto por avanços recentes nesta área, quanto pela natureza do problema de controle. O pêndulo invertido já foi controlado com diversos métodos clássicos e, apesar de sua simplicidade, muitos destes métodos não apresentam um controle robusto neste sistema (BOUBAKER, 2013). Porém, novas técnicas de IA abrem espaço para pesquisas e novos métodos de controle.

1.2 REVISÃO BIBLIOGRÁFICA

Devido a popularidade do pêndulo invertido na área de controle, muitos trabalhos utilizam este sistema para apresentar controladores eficientes e com boa performance, aspectos necessários em diversos projetos. Nesta seção apresenta-se alguns destes trabalhos e suas considerações práticas, bem como projetos similares ao aqui proposto.

1.2.1 Técnicas de Aprendizado de Máquina

Em *reinforcement learning*, o controle é realizado por um agente que busca aprender, por tentativa e erro, a política de controle que maximiza um sinal de recompensa cumulativo (IZZO; MÄRTENS; PAN, 2019). Para isso, o desenvolvedor do projeto define o sinal de recompensa de modo a caracterizar o objetivo de controle.

Assim, na aplicação de RL em robôs, torna-se interessante realizar o treinamento no menor tempo possível (HOSOKAWA; NAKANO, 2012). Para esse fim, a eficiência de dados, caracterizada pela quantidade de episódios necessários para que o controlador aprenda a resolver um problema por tentativa e erro, consiste em uma propriedade essencial neste processo de aprendizado (ADAM; LUCIAN; ROBERT, 2011). Esta eficiência torna-se importante pois o treinamento de um controlador em um sistema real pode ser muito custoso e sua modelagem virtual pode apresentar dificuldades e erros (STEINDÓR; HOFMANN; DEISENROTH, 2018).

No caso do pêndulo invertido, cada episódio finaliza quando a inclinação ou a posição do robô excedem limites pré-estipulados. Por consequência, o robô deve ser deslocado para um novo estado inicial válido, de modo que um novo episódio possa ser iniciado. Assim, o aprendizado ocorre por meio de informações adquiridas em múltiplos episódios sucessivos.

Além disso, um algoritmo eficiente do ponto de vista computacional pode minimizar as durações dos treinamentos por simulações, bem como seus requisitos de hardware e software para implementação prática (ADAM; LUCIAN; ROBERT, 2011).

Com interesse nestas eficiências, Adam, Lucian e Robert (2011) demonstram a utilização da técnica *experience replay* para que os dados de cada iteração, do agente com o sistema, contribuam múltiplas vezes com o aprendizado do controlador de um pêndulo invertido. Este método foi aplicado em ambos os algoritmos empregados no

presente projeto.

Em outro trabalho, Kim *et al.* (2019) utilizaram *imitation learning* para que um algoritmo de RL iniciasse o treinamento com o auxílio de um controlador clássico *Proportional-Integral-Derivative* (PID). Neste caso, o algoritmo finalizou o treinamento sem este auxílio e superou o controlador PID com poucas iterações.

Em outros projetos que misturam controle clássico com IA, métodos de RL foram aplicados para auxiliar o desenvolvimento de controladores clássicos. Kumar *et al.* (2012) utilizaram RL para definir parâmetros de um controlador *fuzzy*. De forma similar, Puriel-Gil, Yu e Sossa (2018) utilizaram *Q-Learning*, método de RL, para configurar um controlador *Proportional-Derivative* (PD).

1.2.2 Trabalhos Similares

A plataforma de código livre *OpenAI Gym* é popular por disponibilizar modelos de sistemas físicos para testes de algoritmos de RL. Entre eles, o modelo de robô do tipo pêndulo invertido tem sido bastante utilizado por trabalhos similares. Porém, este modelo emprega diversas simplificações e limita o sinal de controle em apenas dois valores. Em vista disso, trabalhos mais focados em aplicações práticas desenvolvem modelos mais complexos (LI; LIU; WANG, 2019).

Entre os trabalhos que aplicam métodos de controle clássico no pêndulo invertido, frequentemente se utiliza o modelo linearizado descrito por Ogata (2010). No entanto, métodos de RL não dependem da linearidade do sistema e por consequência podem utilizar modelos não lineares mais complexos.

Sabe-se que imperfeições do modelo podem causar resultados inesperados ao aplicar o controlador em um sistema real (MORIMOTO; DOYA, 2005). Portanto, o desenvolvimento de um modelo para o robô constitui uma parte fundamental deste trabalho.

Pati (2014) modelou de diversas formas um sistema de pêndulo invertido sobre um carrinho. Em particular, seu modelo obtido por equações de Euler-Lagrange é similar ao apresentado no presente projeto. Porém, diferente deste trabalho, seu modelo não considerou o motor que desloca o pêndulo.

Bonarini *et al.* (2008), por sua vez, apresentaram outro trabalho similar, um robô com controle por RL capaz de, além de apenas manter a inclinação do pêndulo nula, variar esta inclinação conforme um sinal de entrada. Seu trabalho utilizou o sensor acelerômetro e, por comparação do autor, obteve-se um controle mais rápido do que o controle clássico *Linear Quadratic Regulator* (LQR).

Em outra pesquisa, Takei, Imamura e Yuta (2009) apresentaram um robô do tipo pêndulo invertido capaz de transportar bagagens por trajetórias pré-definidas. Ficou demonstrado que, em comparação com um robô similar de três rodas, é possível obter um controle mais robusto em apenas duas rodas.

Em um trabalho recente de Mellatshahi *et al.* (2021), foi realizado o controle virtual de um sistema do tipo pêndulo invertido com 6 graus de liberdade. Os autores consideraram o algoritmo *Deep Q-Network* (DQN), também empregado no presente projeto, como estado da arte em *reinforcement learning*.

Bates (2021) apresentou um trabalho recente similar ao presente experimento, em que um algoritmo de RL do tipo *policy gradient* foi treinado de modo virtual no ambiente *OpenAI Gym* e depois aplicado em um robô do tipo pêndulo invertido. Assim, foi obtido sucesso ao equilibrar um pêndulo real. Entretanto, apesar de a região de movimentação do dispositivo ser mantida limitada, a posição exata do robô não foi controlada.

Muitos controladores de pêndulo invertido por meio de RL empregam uma função de recompensa binária, como demonstrado por Bates (2021), que apenas verifica se a inclinação e a posição do pêndulo excederam seus limites dentro de um episódio de simulação. Porém, Mellatshahi *et al.* (2021) aplicaram uma função contínua que considerava o valor da inclinação do dispositivo em cada iteração. Assim, pôde-se obter um desempenho mais eficaz e ainda facilitar a convergência do algoritmo. Entretanto, apesar de uma boa performance no controle da inclinação, a posição do robô não foi controlada a partir de recompensas contínuas.

O presente projeto diferencia-se dos trabalhos similares apresentados aqui, principalmente, devido ao modelo não linear do robô que inclui diversos parâmetros da estrutura mecânica do dispositivo e seu motor.

1.3 OBJETIVOS

Considerando a importância de veículos do tipo pêndulo invertido para pesquisas e aplicações práticas, espera-se que o método de desenvolvimento, dificuldades encontradas e resultados deste projeto sejam úteis para trabalhos futuros, tanto na área de controle como na área de IA. Em especial, devido ao uso de RL, foi almejado obter um controle com múltiplas vantagens em relação a controles clássicos. Deste modo, foram elaborados os objetivos descritos abaixo.

1.3.1 Objetivo Geral

- Desenvolver um controlador para equilíbrio de robôs com estabilidade em duas rodas utilizando aprendizado por reforço.

1.3.2 Objetivos Específicos

- Modelar um robô do tipo pêndulo invertido para simular a tarefa de controle;
- Desenvolver a função custo para o modelo de aprendizado por reforço incluindo a inclinação e a posição do robô.

1.4 ESTRUTURA DO TRABALHO

O robô, objeto deste trabalho, foi formulado e controlado de maneira virtual. Para isso, foi desenvolvido um modelo deste sistema que possibilitou a sua simulação, conforme descrito no capítulo 2. Em seguida, no capítulo 3, apresenta-se os métodos de RL utilizados e seus fundamentos teóricos. Por fim, as simulações realizadas, seus resultados e considerações para aplicações práticas são analisados no capítulo 4.

2 O SISTEMA PÊNDULO INVERTIDO

O sistema do tipo pêndulo invertido foi estudado a partir de sua representação em espaço de estados, tipicamente empregada em sistemas não lineares (KHALIL, 1996). Deste modo, pôde-se realizar análises de estabilidade e simulações partindo de equações diferenciais. Para isso, foi necessário desenvolver um modelo elétrico para o motor a ser utilizado, bem como um modelo mecânico para as partes do robô acopladas ao motor, compostas pelas rodas e o pêndulo.

Assim, desenvolveu-se um modelo com base em parâmetros fundamentais dos componentes utilizados no projeto, através do qual foi possível estudar os efeitos destes parâmetros no comportamento do sistema. E, se necessário, a construção do robô pode ser modificada de modo a satisfazer requisitos de projeto.

Este trabalho foi baseado em um motor real específico. Porém, visto que parâmetros como dimensões físicas e momentos de inércia podem variar de maneira significativa em diferentes concepções de robôs, estas especificações foram estimadas com maior liberdade, sem base em um design de robô específico.

Atualmente, múltiplos métodos são utilizados para modelar e extrair parâmetros de motores de corrente contínua (CC); estes incluem análise de resposta em frequência, estimação de parâmetros algébricos, mínimos quadrados etc. (AWODA; ALI, 2019). Com o objetivo de aprimorar modelos de aplicações críticas, este tópico tem sido alvo de diversas pesquisas. O presente trabalho utilizou o modelo tradicional linear de segunda ordem, empregado em projetos similares de modo satisfatório (MAHAJAN; DESHPANDE, 2013).

2.1 MOTOR DE CORRENTE CONTÍNUA

O presente projeto utilizou um motor CC de imã permanente como atuador acoplado às rodas. Seu uso foi justificado por seu elevado torque de partida e sua versatilidade, que proporciona compatibilidade com uma grande faixa de requisitos de projeto, como velocidade e potência (AUNG, 2007). Além disso, o motor CC foi bastante utilizado em projetos similares, como o robô do tipo pêndulo invertido para locomoção humana desenvolvido por Younis e Abdelati (2009).

Neste sistema, o sinal de controle foi definido pela tensão nos terminais motor, tipicamente aplicada através de um circuito do tipo ponte-H em conjunto com modulação de largura de pulso (PWM, do inglês *Pulse Width Modulation*) (PRIYANKA; MARIYAMMAL, 2018).

2.1.1 Modelo Elétrico

O funcionamento do motor CC é caracterizado pela produção de torque a partir da interação entre o campo magnético de seu imã permanente e a corrente aplicada. Deste modo, assumindo que o fluxo magnético do imã é constante em sua faixa de operação, a relação entre torque τ e corrente i pôde ser modelada pela equação (1), em que K_t representa a constante do motor definida pela construção do dispositivo (HUGHES; DRURY, 2013).

$$\tau = K_t i \tag{1}$$

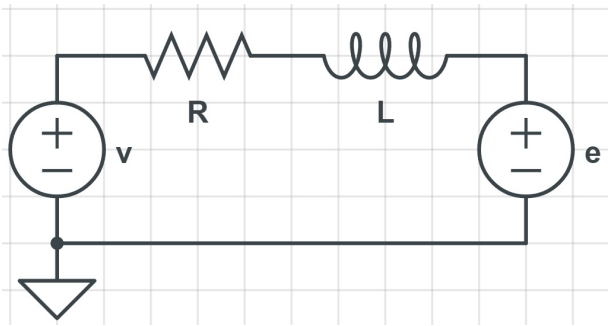
Além disso, o movimento mecânico interno do motor, ao interagir com o fluxo magnético do imã, produz a tensão e denominada força eletromotriz (EMF, do inglês *Electromotive Force*). Desta forma, desconsiderando perdas eletromagnéticas, pôde-se utilizar princípios de conservação de energia para modelar a tensão e em função da constante K_t e da posição angular do eixo do motor φ (AWODA; ALI, 2019).

Porém, nota-se diferenças entre valores medidos para a constante do motor ao analisar as relações de torque com a corrente ou velocidade angular com a EMF (AWODA; ALI, 2019). Um dos fatores que justificam esta desigualdade consiste na desconsideração de perdas na formulação deste modelo. Portanto, utilizou-se a constante do motor como K_e na relação de velocidade angular dada pela equação (2).

$$e = K_e \frac{d\varphi}{dt} \tag{2}$$

Deste modo, a partir da resistência e da indutância de sua armadura, o motor pôde ser modelado pelo circuito da Figura 2, em que v representa a tensão aplicada nos terminais do motor (HUGHES; DRURY, 2013).

Figura 2 – Circuito modelo do motor CC.



Fonte: Elaboração própria.

Desta maneira, o modelo elétrico do motor foi definido pela equação diferencial (3). Nota-se que, neste modelo idealizado, possibilita-se a produção de torque para aumentar a rotação do motor ao aumentar a tensão v. Em contrapartida, um aumento na velocidade de rotação do motor implica em um aumento na tensão e. Deste modo, na

ausência de torque externo, a velocidade de rotação do motor em regime permanente é definida pela tensão v no caso em que a corrente no motor é nula e $e = v$.

$$v = Ri + L\frac{di}{dt} + K_e\frac{d\varphi}{dt} \tag{3}$$

2.1.2 Função de Transferência

Considerando uma carga com momento de inércia J, neste caso composta pelo eixo do motor e as rodas, e o coeficiente de atrito dinâmico interno do motor b, o movimento de rotação do motor pôde ser modelado pela equação (4) (MAHAJAN; DESHPANDE, 2013).

$$J\frac{d^2\varphi}{dt^2} = \tau - b\frac{d\varphi}{dt} \tag{4}$$

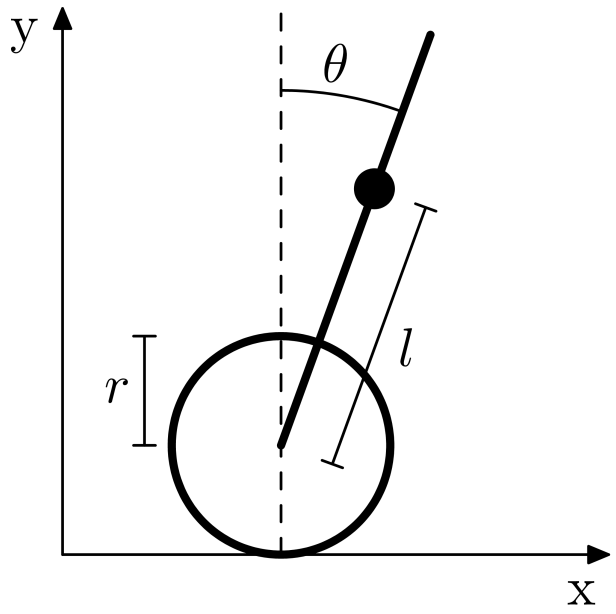
Neste caso, na ausência de torque externo, as equações (1), (3) e (4) foram utilizadas para obter a função de transferência da equação (5).

$$\frac{\dot{\Phi}(s)}{V(s)} = \frac{K_t}{(R + Ls)(b + Js) + K_tK_e} \tag{5}$$

Deste modo, na ausência de torque externo, nota-se que o motor pôde ser modelado por um sistema estável de segunda ordem, contendo um polo de origem mecânica e um polo de origem elétrica.

2.2 MODELO MECÂNICO

Figura 3 – Diagrama do robô do tipo pêndulo invertido.



Fonte: Elaboração própria.

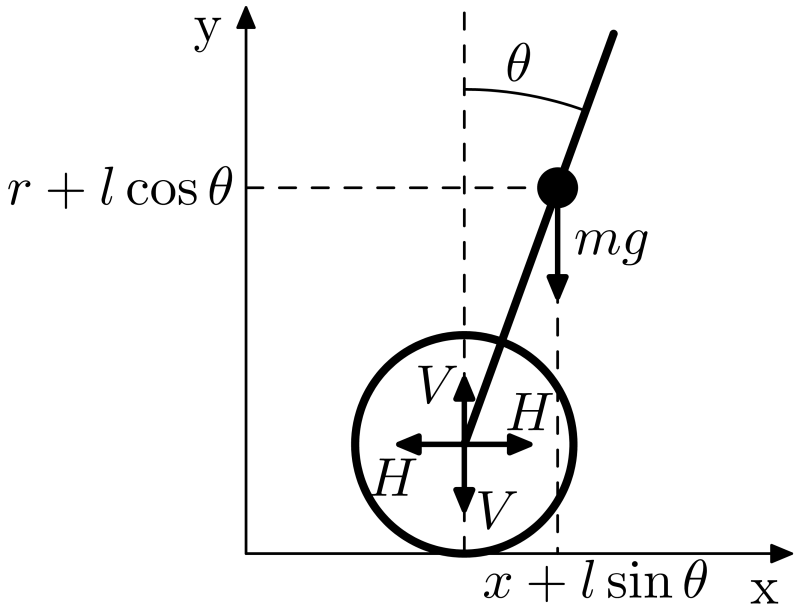
A estrutura mecânica do robô foi representada pelo diagrama da Figura 3, com duas rodas paralelas e um motor. O pêndulo invertido possui inclinação θ e o seu centro de massa está a uma distância l do eixo das rodas de raio r.

2.2.1 Diagrama de Corpo Livre

Os movimentos de rotação e translação das partes deste sistema foram modelados de acordo com as leis de Newton, de modo similar ao método utilizado por Ogata (2010) em um sistema de pêndulo invertido sobre um carrinho. Efeitos de aerodinâmica e deformações em materiais foram desconsiderados. Além disso, as rodas foram consideras sempre em contato com o solo, sem deslizar.

A Figura 4 apresenta o diagrama de corpo livre do sistema com os efeitos da força devido à aceleração de queda livre g que atua sobre o pêndulo de massa m. Deste modo, devido à conexão entre o pêndulo e o eixo das rodas, ambos os corpos percebem forças resultantes de mesmo módulo e direções opostas. Estas forças foram separadas nas componentes de módulos H e V para as direções horizontais e verticais, respectivamente.

Figura 4 – Diagrama de corpo livre.



Fonte: Elaboração própria.

2.2.2 Movimentos de Translação

Em decorrência das forças exibidas no diagrama de corpo livre, o movimento de translação horizontal do pêndulo foi modelado pela equação (6).

$$H = m\frac{d^2}{dt^2}(x + l\sin\theta) \tag{6}$$

De maneira similar, o movimento de translação vertical do pêndulo foi representado pela equação (7), que pôde ser simplificada para a equação (8).

$$V - mg = m\frac{d^2}{dt^2}(r + l\cos\theta) \tag{7}$$

$$V = ml\frac{d^2}{dt^2}\cos\theta + mg \tag{8}$$

2.2.3 Movimentos de Rotação

O movimento de rotação do pêndulo depende de seu momento de inercia I e do torque aplicado pelas componentes de força vertical e horizontal conforme a equação (9).

$$I\frac{d^2\theta}{dt^2} = Vl\sin\theta - Hl\cos\theta \tag{9}$$

De modo semelhante, o movimento de rotação das rodas foi modelado pela equação (10), composta a partir da equação de rotação do motor (4) e do termo de torque gerado pela componente de força horizontal H.

$$J\frac{d^2\varphi}{dt^2} = \tau - b\frac{d\varphi}{dt} - rH \tag{10}$$

2.2.4 Posição do Robô

A posição do robô x foi definida pela posição do centro das rodas, obtida a partir da posição angular do motor pela relação $x = r\varphi$. Assim, tendo o torque do motor dado pela equação (1), a equação (10) pôde ser escrita em função da posição x e da corrente i conforme a equação (11).

$$J\frac{d^2x}{dt^2} = rK_t i - b\frac{dx}{dt} - r^2H \tag{11}$$

Caso cada uma das duas rodas idênticas utilizasse motores independentes, porém também idênticos, e a tensão de entrada de ambos os motores fosse a mesma, a força H seria distribuída igualmente entre as duas rodas cuja rotação passaria a ser modelada pela equação (12).

$$J\frac{d^2x}{dt^2} = rK_t i - b\frac{dx}{dt} - r^2\frac{H}{2} \tag{12}$$

Nota-se que esta equação pode ser escrita como a equação (13). Neste caso, o sistema representado pela equação (13) se torna equivalente ao sistema representado pela equação (11), porém com os parâmetros J, K_t e b multiplicados por um fator de 2. Portanto, o presente modelo pode ser generalizado para um projeto com dois motores.

$$2J\frac{d^2x}{dt^2} = r2K_t i - 2b\frac{dx}{dt} - r^2H \tag{13}$$

2.3 REPRESENTAÇÃO EM ESPAÇO DE ESTADOS

Com o objetivo de analisar e simular o sistema modelado, este foi representado em um espaço de estados (KHALIL, 1996). Deste modo, o vetor de estado s foi formado pelas variáveis de estado $s_1 = \theta$, $s_2 = \dot{\theta}$, $s_3 = x$, $s_4 = \dot{x}$ e $s_5 = i$. Para isso, foi necessário obter as equações de estado, ou seja, as equações diferenciais de primeira ordem que fornecem as variáveis de estado.

2.3.1 Equações de Estado

As equações de estado $\dot{s_1} = s_2$ e $\dot{s_3} = s_4$ foram obtidas diretamente, enquanto as demais equações diferenciais foram determinadas a partir das equações desenvolvidas no modelo do sistema.

As equações (6) e (8) foram utilizadas para substituir os termos H e V nas equações (9) e (11). Deste modo, com algumas manipulações algébricas, obteve-se as equações (14) e (15).

$$J\frac{d^2x}{dt^2} = rK_t i - b\frac{dx}{dt} - mr^2\frac{d^2x}{dt^2} - mr^2l\frac{d^2\theta}{dt^2}\cos\theta + mr^2l\left(\frac{d\theta}{dt}\right)^2\sin\theta \tag{14}$$

$$I\frac{d^2\theta}{dt^2} = mgl\sin\theta - ml\cos\theta\frac{d^2x}{dt^2} - ml^2\frac{d^2\theta}{dt^2} \tag{15}$$

Assim, pôde-se utilizar as equações (14) e (15) para obter $\dot{s_2}$ e $\dot{s_4}$. Por fim, a equação (3) foi utilizada para obter $\dot{s_5}$. Deste modo, o sistema foi modelado pelas equações de estado (16) a (20).

$$\dot{s_1} = \dot{\theta} \tag{16}$$

$$\dot{s_2} = \frac{(J+mr^2)mgl\sin\theta - ml\cos\theta(K_t ir - b\dot{x} + mr^2l\dot{\theta}^2\sin\theta)}{(I+ml^2)(J+mr^2) - m^2r^2l^2\cos^2\theta} \tag{17}$$

$$\dot{s_3} = \dot{x} \tag{18}$$

$$\dot{s_4} = \frac{-m^2r^2l^2g\sin\theta\cos\theta + (I+ml^2)(K_t ir - b\dot{x} + mr^2l\dot{\theta}^2\sin\theta)}{(I+ml^2)(J+mr^2) - m^2r^2l^2\cos^2\theta} \tag{19}$$

$$\dot{s_5} = \frac{vr - Rri - K_e\dot{x}}{Lr} \tag{20}$$

2.3.2 Aproximação por Indutância Nula

Nas próximas seções é demonstrado que a indutância do motor L pode ser desconsiderada. Neste caso, a corrente i pôde ser obtida diretamente a partir da tensão no motor v e a variável de estado $\dot{x}$. Para isso, a equação (3) foi utilizada para obter a equação (21). Deste modo, o sistema pôde ser simplificado para apenas quatro variáveis de estado, sem a necessidade de s_5.

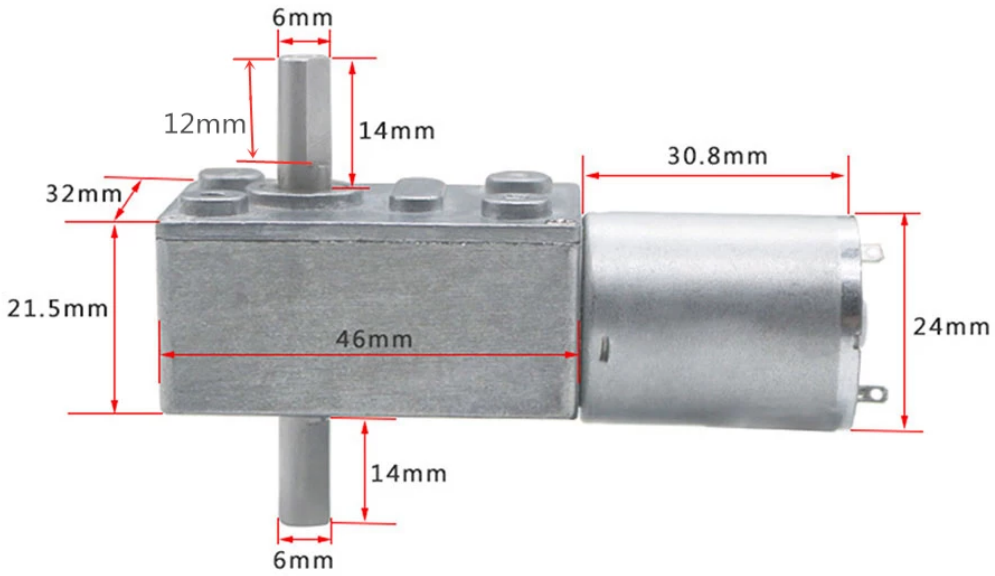
$$i = \frac{rv - K_e\dot{x}}{rR} \tag{21}$$

2.4 EXTRAÇÃO DE PARÂMETROS

Para este projeto, foi idealizado um robô com tamanho adequado para carregar pequenos objetos e dispositivos eletrônicos. Assim, estimou-se o uso de rodas de raio de 2 cm e um pêndulo de 500 g com o centro de massa a 10 cm do centro das rodas.

A partir destes valores, adotou-se o motor exibido na Figura 5, com tensão nominal de 12 V e torque nominal de 0,15 Nm. Sua velocidade nominal de 150 rpm corresponde à velocidade de deslocamento do robô de aproximadamente 0,31 m/s.

Figura 5 – Motor selecionado.



Fonte: Bringsmart[2].

Um único sistema motoredutor de eixo duplo foi utilizado para mover ambas as rodas, o que é suficiente para solucionar o problema de equilíbrio proposto. Assim, o controlador desenvolvido pode servir como base para uma aplicação prática com dois motores de acordo com a equivalência demonstrada na equação (13).

[2] Disponível em: pt.aliexpress.com/item/32889622913.html. Acesso em: 17 jun. 2022.

2.4.1 Caixa de Redução Mecânica

Conforme exibido na Figura 5 da página anterior, o motor possui uma caixa de redução para produzir um torque maior em troca de uma velocidade menor. Deste modo, este efeito foi modelado pelo aumento da constante de torque K_t e a diminuição da constante de velocidade K_e, verificado pelas equações (1) e (2). Porém, devido a perdas de energia causadas pela redução mecânica (FERNANDES *et al.*, 2015), o fator de redução não corresponde exatamente ao aumento percebido em K_t.

Além destes efeitos, espera-se um aumento no fator de atrito b e no momento de inércia J, bem como a introdução de não linearidades ausentes no modelo do motor. As engrenagens da caixa de redução criam uma zona morta de movimento, que ocorre apenas na troca do sentido de rotação (KARA; EKER, 2004). Também se presume um aumento na força de atrito de Coulomb, de módulo aproximadamente constante e em direção oposta ao sentido de rotação (GARCIA; SANTOS; WIT, 2002).

2.4.2 Parâmetros Elétricos do Motor

Os parâmetros elétricos de resistência R e indutância L do condutor do motor podem ser estimados através da aplicação de tensões contínuas e alternadas, respectivamente, e medidas das correntes resultantes (KRISHNAN, 2001). Estes métodos foram empregados pelo instrumento LCR, dispositivo capaz de medir tensão, indutância e outras grandezas de natureza elétrica.

Para isso, foi necessário que a rotação do motor fosse mantida nula, através da aplicação de tensões de baixa amplitude e fixação do eixo do motor. Assim, garantiu-se que nenhum movimento mecânico pudesse gerar EMF, o que poderia interferir nos resultados das medições.

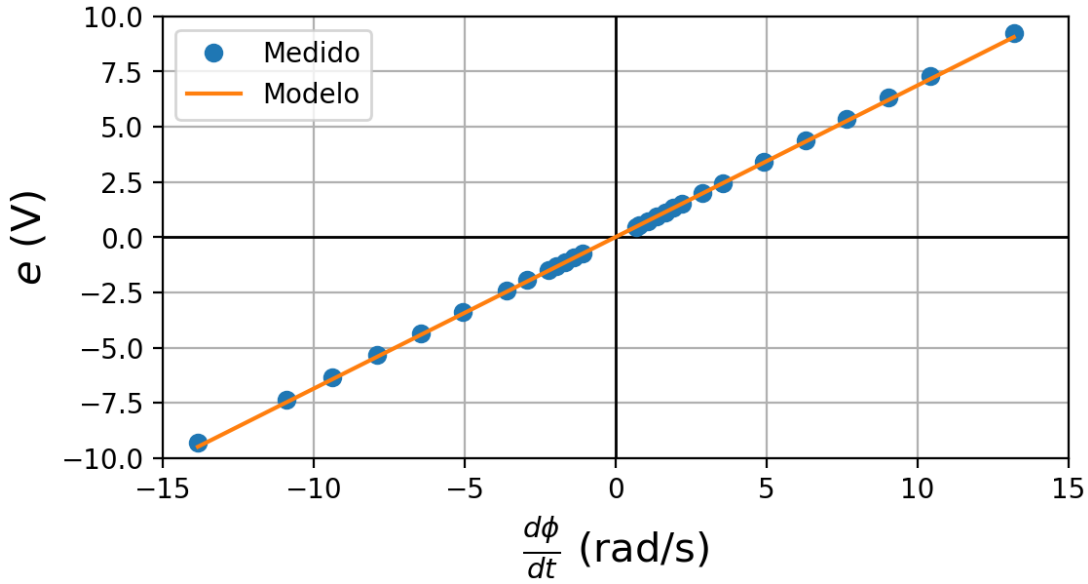
2.4.3 Constante de Velocidade

A constante do motor K_e foi medida pela relação entre a velocidade de rotação do motor e a EMF gerada, conforme a equação (2) (KRISHNAN, 2001). Para isso, foi necessário conhecer a resistência R e calcular a tensão e a partir da tensão aplicada no motor v e a corrente resultante i conforme a equação (22), obtida pela equação (3) em regime estacionário.

$$e = v - Ri \tag{22}$$

Este método foi utilizado para estimar o parâmetro K_e a partir de uma regressão linear com diversas medidas. Deste modo, obteve-se um resultado satisfatório para toda a faixa de operação do motor, conforme apresentado na Figura 6.

Figura 6 – Medida de EMF em função da velocidade angular.



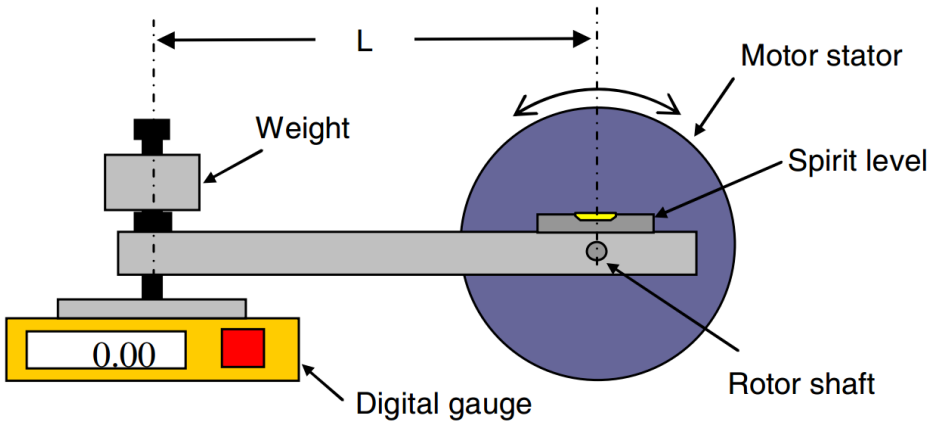
Fonte: Elaboração própria.

2.4.4 Constante de Torque

De modo similar, o parâmetro K_t foi medido pela relação entre torque e corrente. Para isso, o motor foi mantido em regime estacionário com velocidade de rotação nula. Deste modo, pôde-se garantir que não houve interferência na medida de torque devido a forças de atrito dinâmico.

Estimou-se o torque a partir da força tangencial exercida pelo eixo do motor, que pôde ser medida com uma balança e um braço de torque, conforme demonstrado por Zhu (2009) com a aplicação do sistema da Figura 7. Para isso, foi construída uma estrutura de madeira para fixar tanto o motor quanto a balança.

Figura 7 – Sistema de medição de torque.

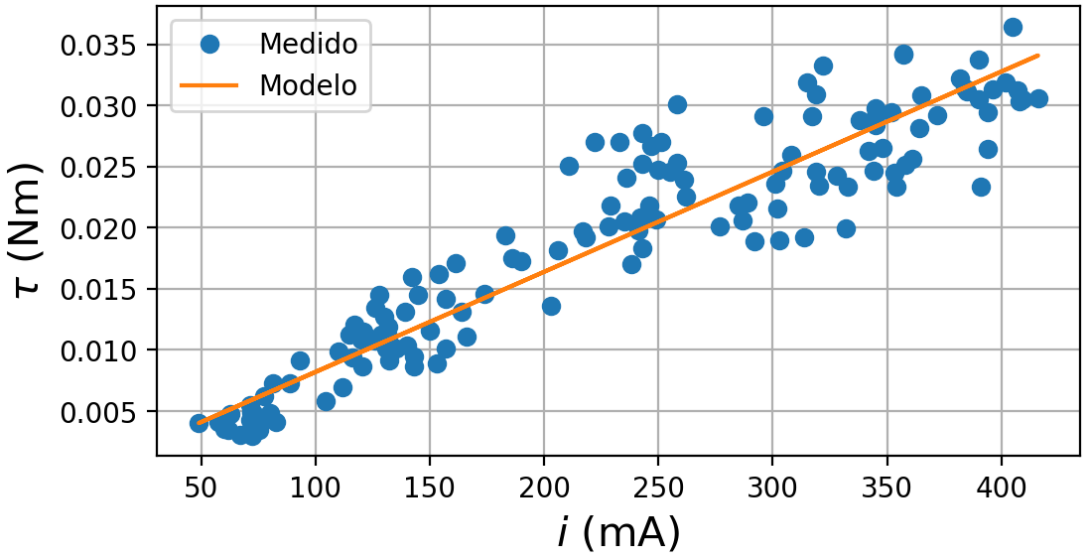


Fonte: Zhu (2009).

O resultado das medidas de torque e o modelo de regressão linear são exibidos no gráfico da Figura 8. Nota-se uma grande variância nos dados, que foi justificada

principalmente por não linearidades no sistema de redução mecânica e no motor. Garcia, Santos e Wit (2002) demonstram que o torque devido a forças de atrito varia em função da posição e da velocidade de rotação do motor. Portanto, a estimativa de K_t obtida foi considerada aceitável para o presente modelo.

Figura 8 – Medida de torque em função da corrente.



Fonte: Elaboração própria.

2.4.5 Coeficiente de Atrito

Após a determinação da constante K_t, o coeficiente de atrito dinâmico b foi estimado através da equação (23), obtida a partir da equação (4) com a velocidade de rotação do motor constante (AUNG, 2007).

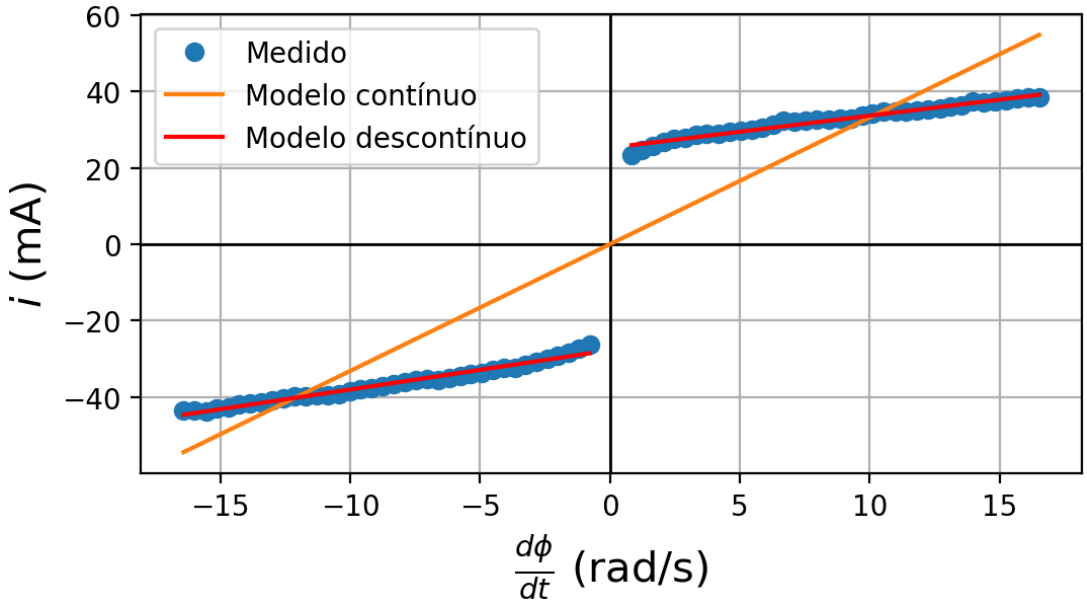
$$b = \frac{K_t i}{\frac{d\varphi}{dt}} \tag{23}$$

Para isso, mediu-se a corrente média consumida pelo motor em diferentes velocidades, em ambos os sentidos de rotação. O resultado das medições, em conjunto com os modelos contínuos e descontínuos elaborados por regressão linear, são exibidos na Figura 9.

O modelo descontínuo indica um atrito estático significativo e coeficientes de atrito dinâmico de 69 μNms e 84 μNms, considerando os dois sentidos de rotação. Por outro lado, o modelo contínuo aproxima estes valores por 272 μNms para toda a região de operação.

Apesar de o modelo descontínuo se adaptar melhor aos dados, este trabalho utilizou o modelo de atrito linear. Deste modo, as simulações a partir de equações diferenciais e as análises de estabilidade são simplificadas. Se necessário, o presente modelo pode ser aprimorado ao considerar estas não linearidades.

Figura 9 – Medida de corrente em função da velocidade angular.



Fonte: Elaboração própria.

Além disso, nota-se que a inclinação da reta do modelo contínuo, obtida por regressão linear, depende da faixa de valores velocidade do motor em que as medidas foram realizadas. Porém, a faixa de operação da velocidade do motor é desconhecida. Além disso, demonstra-se nas próximas seções que, se comparado com os demais parâmetros medidos, o coeficiente de atrito é menos significativo para o comportamento do sistema.

2.4.6 Momentos de Inércia

Por fim, os parâmetros de momento de inércia foram estimados sem a necessidade de medidas. Para isso, o pêndulo foi modelado por uma barra delgada, assim como feito por Sharma (2020) em um sistema do tipo pêndulo invertido semelhante. Deste modo, o momento de inércia I foi definido pela equação (24).

$$I = \frac{ml^2}{3} \tag{24}$$

De maneira similar, para o cálculo do momento de inércia J, as partes do motor e as rodas foram aproximadas por duas rodas cilíndricas maciças de massa M e raio r. Desta forma, obteve-se J pela equação (25).

$$J = Mr^2 \tag{25}$$

2.4.7 Parâmetros Utilizados

Como resultado das medidas e estimações descritas nesta seção, apresenta-se no Quadro 1 os parâmetros de projeto utilizados nas análises e simulações do presente trabalho.

Quadro 1 – Parâmetros do robô.

Raio das rodas ($\mathbf{r}$)	2 cm
Massa do pêndulo ($\mathbf{m}$)	500 g
Massa de cada roda ($\mathbf{M}$)	50 g
Distância entre o centro de massa do pêndulo e o eixo das rodas ($\mathbf{l}$)	10 cm
Resistência do motor ($\mathbf{R}$)	7,4 Ω
Indutância do motor ($\mathbf{L}$)	7,45 mH
Constante de torque do motor considerando a caixa de redução ($\mathbf{K_t}$)	0,082 Nm/A
Constante de velocidade do motor considerando a caixa de redução ($\mathbf{K_e}$)	0,686 Vs
Coeficiente de atrito linear ($\mathbf{b}$)	272 μNms
Momento de inércia do pêndulo ($\mathbf{I}$)	1,67 gm²
Momento de inércia das rodas ($\mathbf{J}$)	0,02 gm²

Fonte: Elaboração própria.

2.5 ANÁLISE DE PONTOS DE EQUILÍBRIO

Devido à não linearidade do sistema em estudo, sua estabilidade foi analisada com base em seus pontos de equilíbrio, que ocorrem em $\dot{\mathbf{s}}(t) = 0$ e $v(t) = 0$ (KHALIL, 1996). Para isso, nota-se pelas equações (16), (18) e (20) que s_2, s_4 e s_5 devem ser nulas. Assim, de acordo com as equações (17) e (19), é necessário que $\sin x_1 = 0$ para que $\dot{\mathbf{s}}$ seja nulo. Portanto, os pontos de equilíbrio ocorrem em qualquer valor de posição x desde que a inclinação θ seja múltipla de π e os demais estados sejam nulos.

Visto que o modelo não considera uma colisão do pêndulo com o chão, pois interrompe-se a simulação antes disso ocorrer, o sistema modelado pode representar o pêndulo não invertido. Deste modo, os pontos de equilíbrio obtidos estão de acordo com o esperado, pois nestes valores o pêndulo se encontra em repouso na posição vertical, invertido ou não invertido. O ponto de equilíbrio de interesse ocorre em $\mathbf{s} = 0$, em que o pêndulo se encontra invertido.

2.5.1 Linearização

O sistema não linear caracterizado pela equação de estado (26) foi representado por sua forma linear definida pela equação (27), válida em seus pontos de equilíbrio.

Deste modo, esta aproximação possibilitou a análise da estabilidade do sistema não linear em torno de seus ponto de equilíbrio.

$$\dot{\mathbf{s}} = \begin{bmatrix} f_1(\mathbf{s}) & f_2(\mathbf{s}) & f_3(\mathbf{s}) & f_4(\mathbf{s}) & f_5(\mathbf{s}) \end{bmatrix}^T \tag{26}$$

$$\dot{\mathbf{s}} = \mathbf{A}\mathbf{s} \tag{27}$$

Para o pêndulo invertido, $\mathbf{A}$ consiste na matriz jacobiana avaliada em $\mathbf{s} = 0$. Esta foi obtida através da equação (28) e apresentada na equação (29).

$$\mathbf{A} = \begin{bmatrix} \frac{df_1}{ds_1} & \frac{df_1}{ds_2} & \frac{df_1}{ds_3} & \frac{ds_1}{ds_4} & \frac{df_1}{ds_5} \\ \frac{df_2}{ds_1} & \frac{df_2}{ds_2} & \frac{df_2}{ds_3} & \frac{ds_2}{ds_4} & \frac{df_2}{ds_5} \\ \frac{df_3}{ds_1} & \frac{df_3}{ds_2} & \frac{df_3}{ds_3} & \frac{ds_3}{ds_4} & \frac{df_3}{ds_5} \\ \frac{df_4}{ds_1} & \frac{df_4}{ds_2} & \frac{df_4}{ds_3} & \frac{ds_4}{ds_4} & \frac{df_4}{ds_5} \\ \frac{df_5}{ds_1} & \frac{df_5}{ds_2} & \frac{df_5}{ds_3} & \frac{ds_5}{ds_4} & \frac{df_5}{ds_5} \end{bmatrix}\Bigg|_{\mathbf{s}=0} \tag{28}$$

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ \frac{(J+r^2m)mgl}{(I+ml^2)(J+mr^2)-r^2m^2l^2} & 0 & 0 & \frac{lmb}{(I+ml^2)(J+mr^2)-r^2m^2l^2} & \frac{-lmK_tr}{(I+ml^2)(J+mr^2)-r^2m^2l^2} \\ 0 & 0 & 0 & 1 & 0 \\ \frac{-r^2m^2l^2g}{(I+ml^2)(J+mr^2)-r^2m^2l^2} & 0 & 0 & \frac{-(I+ml^2)b}{(I+ml^2)(J+mr^2)-r^2m^2l^2} & \frac{(I+ml^2)K_tr}{(I+ml^2)(J+mr^2)-r^2m^2l^2} \\ 0 & 0 & 0 & -\frac{K_e}{Lr} & -\frac{R}{L} \end{bmatrix} \tag{29}$$

Nota-se que a terceira coluna de $\mathbf{A}$ é nula, pois $\dot{\mathbf{s}}$ não depende de x. Isto indica que o sistema pode ser simplificado para uma representação independente de x, com apenas quatro variáveis de estados. Porém, esta variável foi mantida por ser útil para a simulação da tarefa de controle.

Com a finalidade de compreender melhor o sistema em estudo e validar seus resultados, o mesmo procedimento de linearização foi aplicado para o ponto de equilíbrio em que $\theta = \pi$. Desta forma, o sistema foi analisado com base nos autovalores de $\mathbf{A}$, apresentados na Tabela 1.

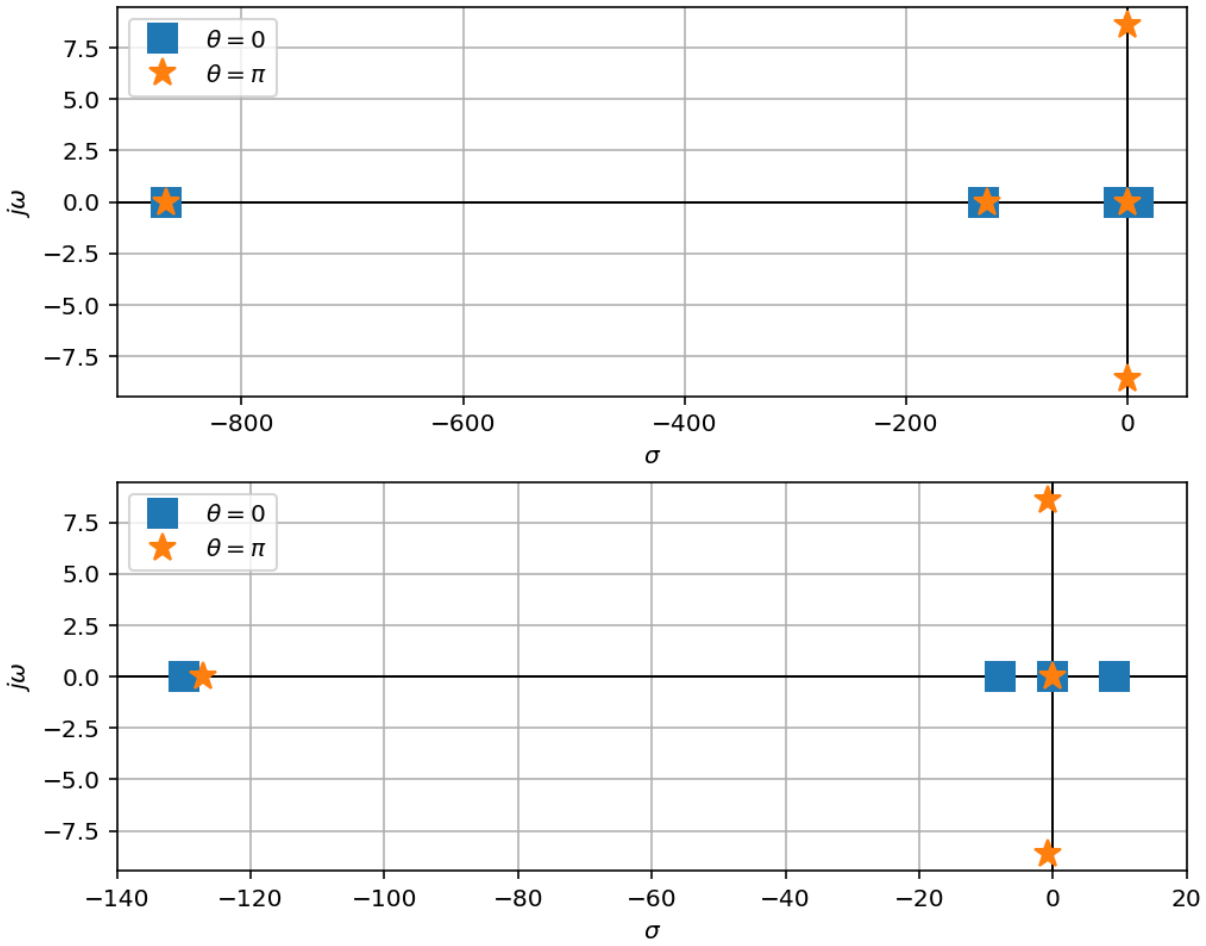
Tabela 1 – Autovalores de $\mathbf{A}$.

	$\boldsymbol{\theta = 0}$	$\boldsymbol{\theta = \pi}$
a_1	0	0
a_2	$9{,}25$	$-0{,}70 + 8{,}59j$
a_3	$-7{,}86$	$-0{,}70 - 8{,}59j$
a_4	-130	-127
a_5	-869	-869

Fonte: Elaboração própria.

Estes valores também são plotados nos gráficos da Figura 10, em duas escalas diferentes para destacar os valores próximos à origem.

Figura 10 – Autovalores de $\mathbf{A}$.



Fonte: Elaboração própria.

2.5.2 Análise do Sistema Linearizado

Os autovalores na origem são decorrentes da variável de estado x, obtida através da integração da variável de estado $\frac{dx}{dt}$. Assim, estes autovalores podem ser desconsiderados ao estudar o sistema de forma independente da posição do pêndulo x.

Conforme esperado, o pêndulo não invertido é estável e o pêndulo invertido é instável, pois a instabilidade ocorre quando $\mathbf{A}$ possui autovalores com parte real positiva.

Percebeu-se também que o coeficiente de atrito b modela perdas de energia, pois se este parâmetro for considerado nulo, os autovalores a_2 e a_3 são deslocados para o eixo imaginário no caso do pêndulo não invertido. Neste caso, o sistema se

torna marginalmente estável e os estados próximos ao ponto de equilíbrio tendem a oscilar.

Com a indutância L desconsiderada, o sistema passou a não depender da variável de estado i e, portanto, a última linha e a última coluna de $\mathbf{A}$ foram eliminadas. Desta forma, o autovalor a_5, bastante distante da origem, foi extinto e os demais autovalores foram levemente deslocados. Isto indicou que a indutância do motor representa uma dinâmica rápida em comparação com as demais propriedades do sistema. Portanto, a simplificação deste parâmetro foi avaliada através de simulações.

2.6 MÉTODO DE SIMULAÇÃO

O sistema do tipo pêndulo invertido foi simulado por suas equações de estado. Com esta finalidade, métodos numéricos foram empregados para resolver estas equações diferenciais. Assim, a partir de um estado inicial e o sinal de tensão do controlador, obteve-se uma sequência de estados percorridos durante um determinado período de simulação.

2.6.1 Frequência de Operação

O algoritmo de controle atua de modo periódico, em um tempo limitado este deve ser capaz de estimar os estados do sistema a partir de sensores e computar um sinal para ser aplicado ao motor. Deste modo, uma frequência de atuação alta requer alto poder computacional e alta taxa de aquisição para os sensores. Por outro lado, uma frequência de atuação baixa pode comprometer o desempenho do controlador.

Para esta frequência, Younis e Abdelati (2009) consideraram 100 Hz mais do que o suficiente para controlar um robô, do tipo pêndulo invertido para locomoção humana, com o algoritmo PID. Em um projeto similar, Bonarini *et al.* (2008) utilizaram 50Hz para simulações de um robô do tipo pêndulo invertido com RL. Portanto, optou-se por utilizar 100Hz para este trabalho, o que significa um intervalo de atuação de 10 ms.

2.6.2 Métodos Numéricos para Solução de Equações Diferenciais

As simulações e os treinamentos dos algoritmos de RL foram desenvolvidos em *Python*. Para isso, diversos métodos de solução de equações diferenciais disponibilizados pela biblioteca *SciPy* foram experimentados. Por fim, o algoritmo *Livermore Solver for Ordinary Differential Equations* (LSODA) implementado pela função *odeint* foi selecionado. Este método apresentou resultados satisfatórios e mais velozes em comparação com os métodos *Backward Differentiation Formula* (BDF), Radau e Runge-Kutta implementados pela função *solve_ivp*.

Admitiu-se uma tolerância de 10^{-12} para os erros relativos e absolutos das soluções das equações diferenciais. Deste modo, os primeiros testes foram realizados

com estados iniciais aleatórios e sinal de controle nulo ou aleatório.

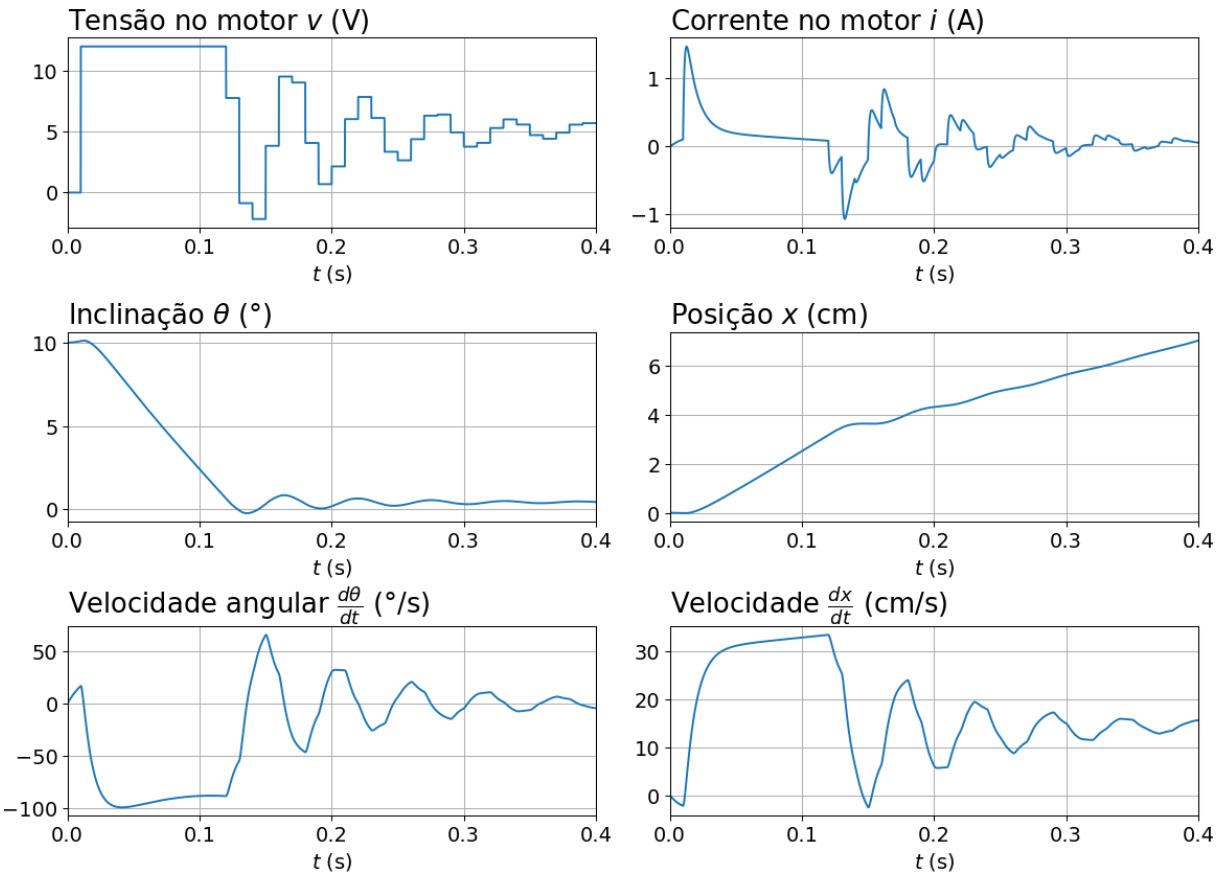
Para simulações em que as variáveis de estado não foram limitadas, os métodos empregados divergiram para alguns estados iniciais. Porém, foi percebido que estas divergências ocorrem apenas para $\theta = \pi$, o que representa o pêndulo não invertido. Portanto, este estado não afeta a tarefa de controle, visto que a simulação é interrompida quando a inclinação do pêndulo passa de determinados limiares.

Ao desconsiderar a indutância do motor, o sistema de equações diferenciais foi simplificado para apenas quatro equações. Neste caso, a velocidade de simulação aumentou para aproximadamente o dobro. Assim, do ponto de vista computacional, esta aproximação contribui com a eficiência de treinamento dos algoritmos de controle.

2.6.3 Análise de Simulações com Controlador Proporcional

Com o objetivo de analisar as simulações e o comportamento das variáveis de estados com um controlador, elaborou-se um simples controlador proporcional. Para isso, apenas a inclinação do pêndulo θ foi controlada e a posição do robô x foi desconsiderada pelo controlador. Deste modo, a partir de uma simulação de 400 ms, obteve-se como resultado os sinais exibidos nos gráficos da Figura 11.

Figura 11 – Simulação com controlador proporcional.



Fonte: Elaboração própria.

Para isso, foi utilizado um ganho de 700 com a inclinação θ interpretada em radianos pelo controlador. O estado inicial foi dado por uma inclinação de 10° e as demais variáveis de estado nulas.

Nota-se que, apesar do sinal de controle v saturar em 12 V no início da simulação, a inclinação foi mantida próximo de zero com sucesso. Porém, principalmente devido ao erro em regime estacionário do controlador proporcional, a inclinação θ foi mantida positiva e portanto a variável x aumentou de forma descontrolada.

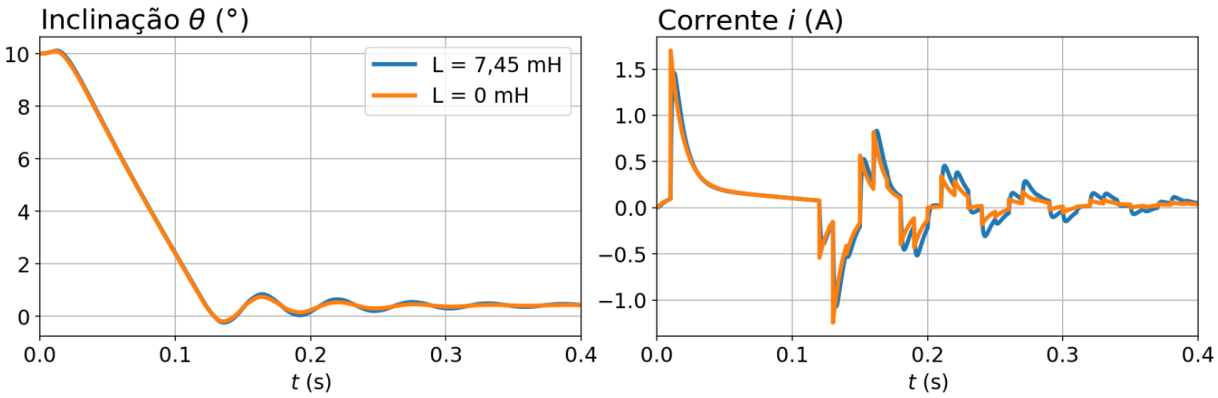
O controlador PID, que generaliza o controlador proporcional, pode ser aplicado de modo a controlar tanto a inclinação quanto a posição do robô, conforme demonstrado por Ponce, Molina e Alvarez (2014). Para isso utilizou-se dois controladores independentes, enquanto um atua em θ, o outro atua em x. Deste modo, o sinal v foi definido pela soma da saída dos dois controladores. Porém, no contexto do presente trabalho, o desenvolvimento destes controladores foge do escopo de análise desta seção.

2.6.4 Análise de Parâmetros e Simplificações

A simulação desenvolvida com o controlador proporcional foi utilizada para analisar efeitos de variações de parâmetros do projeto. Visto que a indutância do motor afeta o tempo de simulação de maneira considerável e a análise de pontos de equilíbrio indicou que este parâmetro pode apresentar pouca influência nas simulações, avaliou-se uma aproximação por indutância nula.

Os gráficos da Figura 12 comparam os resultados da simulação apresentada anteriormente com sua versão aproximada. Nota-se que há pouca diferença nas trajetórias das variáveis de estado. Portanto, esta aproximação foi considerada válida e adotada neste projeto.

Figura 12 – Simulações com aproximação da indutância.

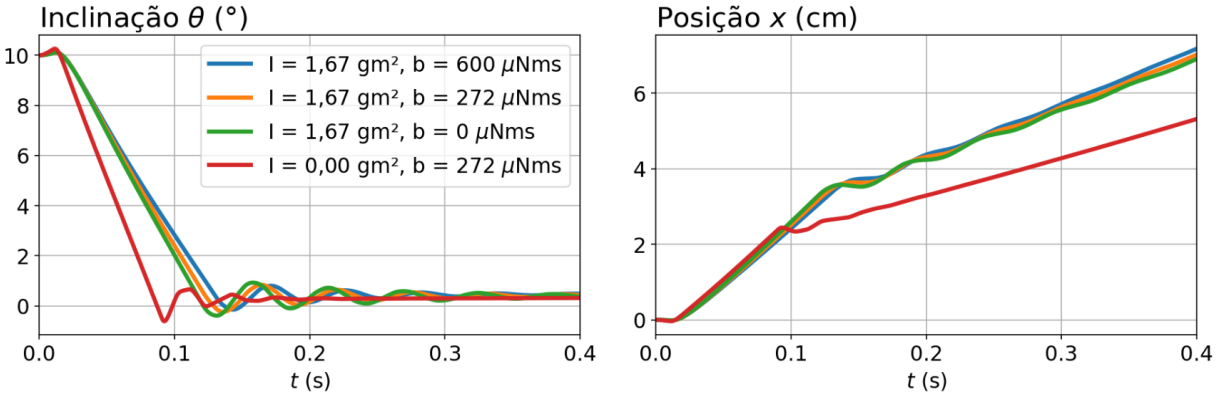


Fonte: Elaboração própria.

Este procedimento de comparação também foi aplicado em outros parâmetros

do projeto. Os gráficos da Figura 13 exibem os efeitos de desconsiderar o coeficiente de atrito b e o momento de inércia do pêndulo I. Nota-se que, em comparação com outros parâmetros, o coeficiente de atrito foi pouco relevante para o resultado das simulações.

Figura 13 – Simulações com aproximações de coeficiente de atrito e momento de inércia.



Fonte: Elaboração própria.

Assim, tolera-se erros maiores na medida do coeficiente de atrito. Porém, outros modelos de força de atrito não foram testados e podem apresentar uma relevância maior.

Por outro lado, a desconsideração do momento de inércia do pêndulo causou uma variação significativa nas simulações. Isto indica que, ao modelar um robô real, a medida deste momento de inércia pode ser essencial. Além disso, a distribuição de massa do robô pode ser ajustada de modo a modificar este parâmetro e facilitar a tarefa de controle.

3 APRENDIZADO POR REFORÇO

Considerando a natureza da tarefa de controle e os resultados de trabalhos similares, decidiu-se aplicar dois algoritmos de RL. O primeiro algoritmo implementado, *Double DQN*, constitui uma variação do método popular *Deep Q-Network*. Estes fornecem um sinal de saída discreto a partir de um sinal de entrada contínuo.

Por outro lado, o segundo algoritmo aplicado, TD3, constitui um algoritmo mais recente com potencial de produzir melhores resultados em relação ao método *Double DQN*. Isto ocorre principalmente devido a seu tipo de sinal de saída, considerado mais adequado para o pêndulo invertido equilibrado através de um motor, pois este utiliza ações em domínio contínuo (BI; CHEN; XIAO, 2021).

3.1 FUNDAMENTOS

Aprendizado por reforço consiste em um tópico de aprendizado de máquina em que um controlador, tipicamente chamado de agente, aprende como solucionar um problema de modo a maximizar um sinal de recompensa r_t (IZZO; MÄRTENS; PAN, 2019). Para isso, este agente desenvolve uma política π que representa sua probabilidade de tomar a ação a dado o estado s.

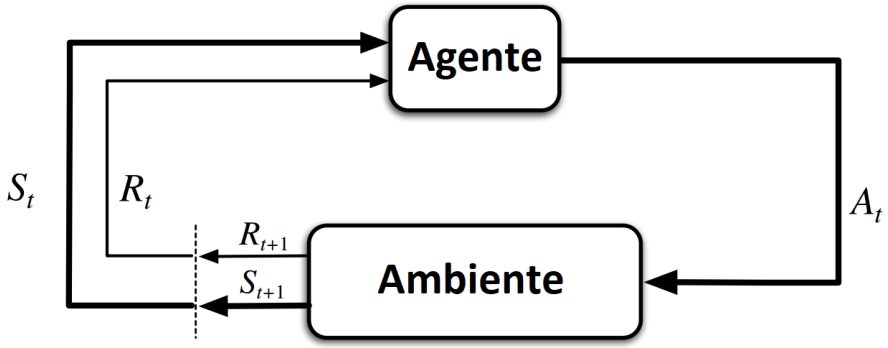
3.1.1 Processo de Decisão de Markov

O modelo de pêndulo invertido desenvolvido compõe um Processo de Decisão de Markov (MDP, do inglês *Markov Decision Process*). Isto ocorre porque os estados do sistema podem ser totalmente definidos em função de suas variáveis de estado. Assim, ao conhecer o estado atual s_t e a ação tomada a_t, torna-se possível obter a probabilidade de ocorrer uma transição para um próximo estado s_{t+1}. Neste sistema de pêndulo invertido, emprega-se um MDP determinístico. Deste modo, pôde-se predizer toda a sua futura trajetória de estados pelo conhecimento do estado inicial e das ações executadas.

Adotou-se a notação de S_t, A_t e R_t para as variáveis aleatórias que podem assumir valores específicos de estado s_t, ação a_t e recompensa r_t, respectivamente, para o instante de tempo discreto t.

No sistema modelado, o estado s composto por variáveis de estado possui a mesma interpretação do estado s do MDP. Assim, para um determinado estado s_t, os agentes treinados aplicaram os sinais de controle definidos pelas políticas $\pi(a_t|S_t = s_t)$ e as simulações do modelo retornaram o próximo estado S_{t+1} em conjunto com o sinal de recompensa R_{t+1}. Repetiu-se este procedimento, representado pela Figura 14, para que os agentes pudessem aprender e aprimorar as suas políticas de modo a maximizar o sinal de recompensa.

Figura 14 – A interação entre agente e ambiente em um MDP.



Fonte: (SUTTON; BARTO, 2018, tradução nossa)

3.1.2 Sinal de Recompensa

Tipicamente, métodos de aprendizado de máquina buscam minimizar uma função custo, também conhecida como função perda. Em vista disso, esta foi definida, pelos métodos aplicados, como o oposto da função recompensa acumulada. O sinal de recompensa caracteriza totalmente o objetivo do controlador, portanto sua definição impacta diretamente a performance do agente, com importância significativa para o projeto de RL (POLYDOROS; NALPANTIDIS, 2017).

Neste trabalho, foram controladas a inclinação do pêndulo θ e a posição do robô x. Assim, o sinal de recompensa r_t foi definido em função de variáveis de estado conforme a equação (30). Deste modo, os coeficientes λ_x e λ_θ definem a importância do controle das variáveis x e θ, respectivamente.

$$r_t = -\lambda_x |x_t| - \lambda_\theta |\theta_t| \tag{30}$$

Desta maneira, com o objetivo de maximizar o sinal de recompensa, o agente busca estados em que $x = 0$ e $\theta = 0$. Portanto, para controlar o robô para uma posição diferente, basta modificar o referencial de medida de x.

Nota-se que, deste modo, o agente poderia escolher uma ação que trouxesse os valores de posição x e inclinação θ mais próximos de 0, mas como consequência aumentaria a velocidade do robô e portanto as recompensas dos próximos estados seriam comprometidas. Além disso, se o sistema se encontrasse em um estado com θ próximo de zero e x distante de zero, o agente poderia escolher manter o robô nesta posição e preservar o sinal de recompensa em um ponto máximo local.

Isto poderia ocorrer pois, para o robô se deslocar de modo a corrigir a posição x, o pêndulo deve ser inclinado de modo que θ degrade a recompensa imediata. Desta forma, para aumentar os sinais de recompensa futuros, as recompensas dos estados próximos devem decair. Este tipo de situação dificulta a tarefa de controle por RL (SUTTON; BARTO, 2018).

Para resolver este problema, considerou-se a recompensa acumulada G_t, dada pela equação (31), em vez do sinal de recompensa imediata R_t.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... \tag{31}$$

Nesta equação, γ representa um fator de desconto com valor entre 0 e 1. Deste modo, γ próximo de zero representa uma recompensa acumulada focada em estados próximos. Por outro lado, γ próximo de 1 demonstra maior interesse em recompensas futuras. Este parâmetro contribui com a qualidade da política obtida, porém pode aumentar o tempo de convergência do algoritmo, visto que o sinal de recompensa acumulada passa a depender de múltiplas transições de estados (POLYDOROS; NALPANTIDIS, 2017).

3.1.3 Função Q

Através das informações obtidas por meio de transições em um MDP, pode-se estimar a recompensa acumulada para um determinado par de estado e ação. Para isso, define-se q_π, que representa o valor esperado de G_t ao seguir a política π, pela equação (32). Para a política ótima, que retorna a melhor ação possível para maximizar o valor q, utiliza-se a notação q_*.

$$q_\pi(s, a) = E_\pi[G_t \mid S_t = s, A_t = a] \tag{32}$$

Visto que a interação do agente com o ambiente fornece apenas a recompensa imediata R_{t+1}, escreve-se a equação (31) de forma recursiva para obter a equação (33).

$$G_t = R_{t+1} + \gamma G_{t+1} \tag{33}$$

Deste modo, substitui-se a equação (33) na equação (32) com a política ótima para obter a equação de Bellman (34).

$$q_*(s, a) = E[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \mid S_t = s, A_t = a] \tag{34}$$

Ao analisar os estados de forma independente, esta equação representa um sistema de equações cujo tamanho depende do número de estados. Deste modo, a partir de sua solução, é possível determinar a política ótima ao escolher, para cada estado, a ação que maximiza o valor q. Entretanto, a solução deste sistema de equações pode apresentar um custo computacional inviável devido ao número de possíveis estados.

Métodos como programação dinâmica buscam a solução destas equações, porém, algoritmos de RL tipicamente são satisfeitos com apenas uma estimativa desta solução (SUTTON; BARTO, 2018). Ambos os métodos aplicados neste trabalho são baseados na estimativa de q_π pelo valor denominado Q_π.

3.2 DOUBLE DEEP Q-NETWORK

O algoritmo *Double Deep Q-Network*, aplicado neste projeto, é baseado no método *Deep Q-Network*, que por sua vez utiliza o método tradicional *Q-Learning* em conjunto com redes neurais para trabalhar com espaços de estados contínuos.

3.2.1 Q-Learning

Q-Learning consiste em um método tabular apresentado por Watkins (1989). Este algoritmo estima $q_*(s, a)$ por $Q(s, a)$ e aprimora esta aproximação através de dados coletados pelo agente ao interagir com o ambiente. Deste modo, ao conhecer o valor Q para cada estado e ação discretos, obtém-se a política ótima ao selecionar a ação que maximiza Q conforme estados encontrados.

A partir de valores iniciais de $Q(S_t, A_t)$, o aprendizado ocorre por meio de iterações com o ambiente que possibilitam novas estimativas definidas por $Q'(S_t, A_t)$. Deste modo, utiliza-se a equação de Bellman (34) para obter $Q'(S_t, A_t)$ conforme a equação (35). Esta técnica de utilização da própria função Q para calcular outra função estimada Q' é conhecida como *bootstrapping*.

$$Q'(S_t, A_t) = R_{t+1} + \gamma \max_a Q(S_{t+1}, a) \tag{35}$$

Nota-se que a função $Q(s, a)$ aprendida se refere à política aplicada durante o treinamento e não à política ótima. Entretanto, à medida que os valores Q se tornam mais precisos, a política aplicada converge para a política ótima. Sendo assim, utiliza-se o método de diferença temporal para aprimorar a estimativa do valor Q por iterações conforme a atualização de parâmetro (36).

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(Q'(S_t, A_t) - Q(S_t, A_t)) \tag{36}$$

O fator α indica a taxa de aprendizado. Ao aumentar seu valor, as atualizações de cada iteração se tornam mais relevantes e a velocidade de aprendizado pode ser aumentada. Porém, visto que $Q(S_t, A_t)$ pode ser uma estimativa ruim, o parâmetro α também aumenta a variância do processo de aprendizagem. Portanto, este parâmetro deve ser ajustado de acordo com a tarefa de controle.

3.2.2 Exploração

Para que a aprendizagem produza resultados satisfatórios, deve-se garantir que este processo encontre amostras de diversas regiões do espaço de estado (NAIR *et al.*, 2018). Com esta finalidade, utiliza-se a política de exploração β, com ações aleatórias, em conjunto com a política aprendida π.

Ações fornecidas pela política π tendem a manter pêndulo invertido equilibrado e portanto coletam amostras de estados com maior probabilidade de serem encontrados

pelo controlador. Além disso, evitar a queda do pêndulo durante o treinamento pode ser interessante para aplicações em robôs reais.

Por outro lado, ações aleatórias podem resultar na exploração de estados e ações que normalmente não seriam encontrados pela política π. Portanto, este processo de exploração contribui com o aprimoramento da política π. Deste modo, a política β é definida pela escolha com probabilidade ε de uma ação aleatória e a escolha de uma ação dada por π com probabilidade $1 - \varepsilon$.

Durante os primeiros episódios de treinamento, a política π se encontra bastante distante da política ótima e, portanto, é pouco relevante para a exploração. Em vista disso, este trabalho aplicou a técnica de decaimento linear de ε em função das iterações do agente com o ambiente. Este parâmetro foi inicializado em $\varepsilon = 1$ e reduzido, durante os treinamentos, até o valor mínimo de 1%.

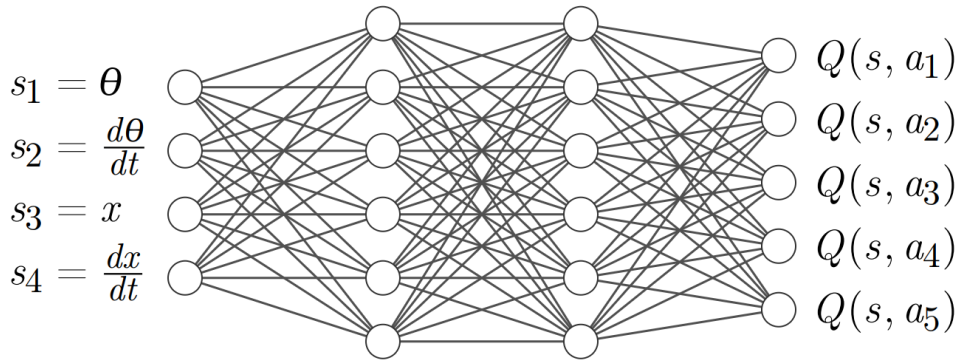
3.2.3 Aproximação por Redes Neurais

O método tabular *Q-Learning* apresenta dificuldades em aplicações com espaços de estados contínuos ou de grandes dimensões. Isto ocorre pois o valor Q deve ser avaliado para cada possível par de estado e ação.

Para solucionar este problema, o algoritmo DQN utiliza uma rede neural artificial como função aproximadora de Q. Redes neurais tem sido muito estudas e aplicadas em múltiplas áreas de IA (ABIODUN *et al.*, 2018). Portanto, sua utilização no método DQN foi auxiliada por diversas técnicas de treinamento já desenvolvidas e bem-conceituadas.

Um exemplo de rede neural com os estados de acordo com este trabalho e as ações discretizadas em cinco valores é exibido na Figura 15. O tamanho da entrada da rede foi definido pela dimensão dos estados e a saída da rede foi composta por um valor Q para cada possível ação a.

Figura 15 – Exemplo de rede neural artificial como função aproximadora de Q.



Fonte: Elaboração própria.

Deste modo, o valor $Q(S_t, A_t)$ estimado pela rede neural é otimizado a partir de valores $Q'(S_t, A_t)$ fornecidos pela interação do agente com o ambiente. Assim,

o processo de aprendizado da rede neural utiliza como função perda o erro médio quadrático apresentado na equação (37).

$$L_{MSE} = (Q'(S_t, A_t) - Q(S_t, A_t))^2 \tag{37}$$

3.2.4 Deep Q-Network

O agente *Deep Q-Network*, também conhecido como *Deep Q-Learning*, foi apresentado por Mnih *et al.* (2015). Este método exibiu ótimos resultados em testes com jogos de Atari e se tornou popular para solucionar outras tarefas, incluindo o equilíbrio do pêndulo invertido.

Para isso, foram utilizadas duas técnicas para reduzir a variância do processo de treinamento das redes neurais aproximadoras de Q. Estas técnicas foram consideradas, pelos autores do método DQN, essenciais para a convergência do algoritmo e, consequentemente, para a sua performance.

A primeira técnica empregada, *experience replay*, foi apresentada por Lin (1992). Neste método, as experiências coletadas pelo agente são armazenadas em memória para uso posterior. Assim, cada atualização do valor Q pode utilizar um conjunto de experiencias passadas, também chamado de *batch*, em vez de apenas a última transição de estados.

Este sistema de atualização por *batch* já é empregado em outras técnicas de IA para se obter diversas vantagens, como redução de variância no treinamento e redução de custo computacional ao paralelizar operações. Deste modo, a perda calculada para um *batch* de tamanho N é dada pela equação (38).

$$L_{MSE} = \frac{1}{N} \sum_{j=1}^{N} (Q'_j(S_t, A_t) - Q_j(S_t, A_t))^2 \tag{38}$$

Cada *batch* é construído a partir de seleções aleatórias de experiências da memória. Quando novas experiências são obtidas e a memória está cheia, as informações mais antigas são descartadas para que novas informações sejam armazenadas. Sendo assim, os tamanhos de memória e *batch* constituem hiperparâmetros ajustados de acordo com o treinamento.

Em RL, atualizações sucessivas a partir de apenas experiências recentes induzem a rede neural a se adaptar a uma região limitada do espaço de estados e, neste processo, o aprendizado passado pode ser perdido. Portanto, a técnica *experience replay* reduz a variância do treinamento ao evitar que a rede neural perca a sua capacidade de adaptação a estados e ações passadas.

Além disso, este método contribui com a eficiência de dados, visto que as informações adquiridas pelo agente podem ser utilizadas múltiplas vezes.

Em outra técnica aplicada pelo agente DQN, separou-se a função Q em duas redes neurais. Assim, a equação (35) pôde ser apresentada na forma da equação (39), em que Q_e e Q_t são definidos por redes neurais distintas.

$$Q'_e(S_t, A_t) = R_{t+1} + \gamma \max_a Q_t(S_{t+1}, a) \tag{39}$$

Visto que a rede Q_t tem como função calcular o valor alvo utilizado no processo de atualização da rede Q_e, Q_t é tipicamente chamada de rede alvo e Q_e é denominada rede de avaliação.

Devido às atualizações da rede de avaliação em cada iteração do treinamento, os parâmetros da rede alvo são mantidos constantes por C iterações. Assim, a variância do treinamento é reduzida pois as atualizações da rede de avaliação passam a depender de uma função com variância menor.

A cada C iterações, a rede alvo é atualizada através da cópia dos parâmetros da rede de avaliação. Assim, C constituí um hiperparâmetro de treinamento. Ao aumentar este valor, a variância do treinamento é reduzida, porém o número de iterações necessárias para finalizar o processo de aprendizagem pode aumentar.

3.2.5 Double Deep Q-Network

O método DQN utiliza uma estimativa do valor Q, dada pela rede alvo, para realizar a atualização da rede de avaliação. Assim, realiza-se um *bootstrapping* de acordo com a equação (39) apresentada anteriormente.

Desta forma, visto que a estimativa do valor Q pode conter erros, ao selecionar a ação que maximiza Q conforme o fator $\max_a Q_t(S_{t+1}, a)$, pode acontecer uma superestimação deste valor. Isto ocorre pois, se $Q(S_t, A_t)$ possui um erro aleatório de mediana zero para cada possível ação, é mais provável que o maior valor Q deste conjunto apresente um erro maior que zero.

Hasselt, Guez e Silver (2016) demonstraram que essa superestimação prejudica o desempenho do controlador e propuseram o método *Double DQN* para minimizar este efeito. Para isso, realizou-se apenas uma pequena modificação no método DQN.

Procurou-se separar a escolha da ação a, presente na equação (39), dos valores fornecidos por pela rede alvo. Com este propósito, a ação a passou a ser selecionada de acordo com o Q_e em vez de Q_t. Assim, a nova equação de atualização foi definida por (40).

$$Q'_e(S_t, A_t) = R_{t+1} + \gamma Q_t(S_{t+1}, argmax_a Q_e(S_{t+1}, a)) \tag{40}$$

Visto que há uma cópia periódica de parâmetros entre as redes, estas não são totalmente independentes. Entretanto, o método empregado foi capaz de redu-

zir a superestimação do valor Q e, consequentemente, aumentar o desempenho do controlador de maneira significativa.

3.3 TWIN DELAYED DEEP DETERMINISTIC POLICY GRADIENT

O agente DQN soluciona o problema de representação de espaços de estados contínuos encontrado pelo método *Q-Learning*. Porém, o espaço de ações ainda é tratado de maneira discreta por este método.

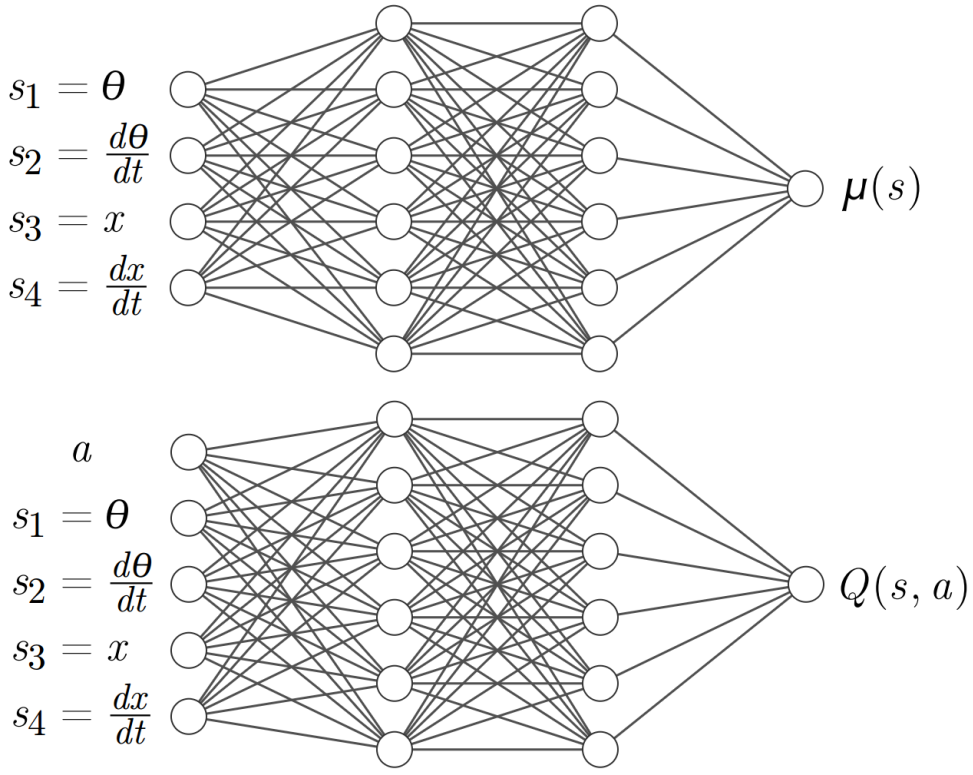
No presente projeto, a discretização da tensão aplicada no motor limita o desempenho do controlador, visto que políticas melhores poderiam ser encontradas a partir de uma resolução de tensão maior. Para solucionar esta dificuldade, aplicou-se o método *Twin Delayed Deep Deterministic Policy Gradient*, também conhecido como TD3, proposto por Fujimoto, Hoof e Meger (2018).

3.3.1 Deep Deterministic Policy Gradient

O método TD3 foi baseado no algoritmo *Deep Deterministic Policy Gradient* (DDPG), apresentado por Lillicrap *et al.* (2016), que por sua vez foi baseado no método *Deterministic Policy Gradient* de Silver *et al.* (2014).

As redes neurais artificiais utilizadas nos métodos DDPG e TD3 são exemplificadas, de acordo com estados e ações do presente trabalho, na Figura 16.

Figura 16 – Exemplo de redes neurais artificiais aplicadas nos métodos DDPG e TD3.



Fonte: Elaboração própria.

Neste caso, além de estimar o valor $Q(s, a)$ pelas redes que fornecem Q_e e Q_t, também foi estimada a ação ótima μ, com valores contínuos, através de uma rede neural artificial.

O processo de treinamento da rede neural que fornece o valor Q é semelhante ao aplicado pelo agente DQN. Neste caso, o valor Q do próximo estado, dado pela rede neural alvo, é calculado a partir da ação fornecida por $\mu(S_{t+1})$ conforme a equação (41).

$$Q'_e(S_t, A_t) = R_{t+1} + \gamma Q_t(S_{t+1}, \mu(S_{t+1})) \tag{41}$$

Por outro lado, o treinamento da rede neural que fornece a ação μ é realizado de acordo com o método *Deterministic Policy Gradient*, do tipo *actor-critic*. Nesta classe de algoritmos, a política que atua de modo a fornecer as ações μ é definida pela rede neural de avaliação. Para que esta rede seja treinada, o valor Q deve ser estimado através das redes que fornecem Q_e e Q_t, denominadas redes neurais críticas. Neste caso, tipicamente se otimiza a política a partir de sua medida de performance J_π, definida pelo valor esperado de Q ao seguir a política π.

Esta performance é calculada em função dos parâmetros Θ que compõe a rede neural atuadora. Assim, as ações μ são otimizadas através do gradiente da performance definido por ∇J_π. Para isso, Silver *et al.* (2014) demonstraram que este gradiente pode ser aproximado pelo gradiente do valor Q ao seguir a política representada pelas ações μ, conforme a equação (42).

$$\nabla_\Theta J_\pi(\Theta) \approx \nabla_\Theta Q(S_t, \mu(S_t, \Theta)) \tag{42}$$

Em vista disso, o valor Q é computado pela rede neural crítica Q_e para obter a perda exibida na equação (43), com um *batch* de tamanho N. Nota-se que o sinal negativo desta equação indica que, ao minimizar a perda, a performance da política é maximizada.

$$L_A = -\frac{1}{N}\sum_{j=1}^{N} Q_{e_j}(S_t, \mu(S_t)) \tag{43}$$

Este algoritmo é classificado como determinístico, pois a política π escolhe ações de forma determinística conforme indicado por μ. Assim, para garantir a exploração do ambiente durante o treinamento, utiliza-se a política exploratória β definida conforme a equação (44), em que η representa um ruído gaussiano com média zero.

$$\beta(s) = \mu(s) + \eta \tag{44}$$

O método DDPG, assim como o método DQN, utiliza a técnica *experience replay* para reduzir a variância do treinamento. Porém, em vez de aplicar atualizações

periódicas na rede Q_e, utiliza-se *Polyak Averaging*. Neste caso, os parâmetros Θ_e da rede Q_e são atualizados conforme os parâmetros Θ_t da rede Q_t em cada iteração pela atualização (45).

$$\Theta_e \leftarrow T\Theta_t + (1-T)\Theta_e \tag{45}$$

T atua como o fator de um filtro passa-baixa e tem o seu valor entre 0 e 1. Assim, este compõe um hiperparâmetro com efeito análogo ao hiperparâmetro C do método DQN. O valor de T é tipicamente definido próximo de zero.

3.3.2 Twin Delayed Deep Deterministic Policy Gradient

Assim como o método *Double DQN* aprimorou o algoritmo DQN ao tratar erros de superestimação do valor Q, o método TD3 demonstrou que estes erros persistem em métodos do tipo *actor-critic* e apresentou soluções baseadas nos algoritmos *Double DQN* e DDPG. Além disso, novas técnicas empregadas no método TD3 colaboraram com a redução da variância de treinamento e portanto contribuem com o desempenho deste algoritmo.

Visto que a rede Q_e empregada no método *Double DQN* tem seus parâmetros copiados de maneira periódica da rede Q_t, estas são bastante similares durante os treinamentos. Assim, o erro de superestimação do valor Q não é totalmente eliminado neste método.

Para solucionar este problema, o método TD3 separa as redes neurais críticas nas quatro redes independentes Q_{e1}, Q_{e2}, Q_{t1} e Q_{t2}. Assim, o valor Q adotado para a atualização das redes de avaliação Q_{e1} e Q_{e2} é definido pelo menor valor Q calculado pelas redes alvo Q_{t1} e Q_{t2}, conforme a equação (46).

$$Q'_{e1,e2}(S_t, A_t) = R_{t+1} + \gamma \min(Q_{t1}(S_{t+1}, \beta_{t+1}), Q_{t2}(S_{t+1}, \beta_{t+1})) \tag{46}$$

Assim como no método DDPG, as redes alvo são atualizadas a partir das redes de avaliação por *Polyak Averaging*. Por outro lado, as redes de avaliação são atualizadas a partir da mesma função perda. Apesar disso, em razão da inicialização aleatória e independente dos parâmetros destas redes neurais, seus valores Q resultantes são diferentes.

Além disso, a rede neural referente à política também é separada em rede alvo μ_t e rede de avaliação μ_e. Assim, as ações β_{t+1} utilizadas pelas redes críticas alvo são definidas pela rede μ_t, enquanto as ações do agente são fornecidas pela rede μ_e. Deste modo, a variância de treinamento é reduzida mais uma vez.

Desta forma, devido à operação que seleciona o valor mínimo entre Q_{t1} e Q_{t2}, podem ocorrer erros de subestimação do valor Q. Entretanto, visto que a política

evita aplicar ações com valores Q pequenos, estes erros não causam tanto dano ao desempenho do agente quanto erros de superestimação.

Da mesma maneira que o método DDPG, TD3 utiliza ações β com ruído aleatório η. Porém, o método TD3 adiciona ruído tanto em ações exploratórias quanto em ações que produzem os valores Q das redes críticas alvo. Deste modo, o ruído atua como uma forma de regularização (BISHOP, 2006), pois entende-se que, para um mesmo estado, ações similares devem gerar valores Q similares.

O método DDPG utiliza o processo de Ornstein-Uhlenbeck para produzir um ruído correlacionado. Porém, demonstra-se pelo método TD3 que o mesmo resultado pode ser obtido com um ruído gaussiano de média zero. Ademais, tanto o ruído quanto as ações com ruído adicionado tem seus valores absolutos limitados. Assim, certifica-se que estas variáveis não podem assumir valores extremos que poderiam comprometer o treinamento.

No início de cada treinamento, as ações fornecidas pela política aprendida estão longe de seus valores ótimos. Portanto, as primeiras D ações escolhidas pelo agente são totalmente aleatórias, de modo a explorar o ambiente sem possíveis limitações causadas pela política.

Além disso, as redes μ_e e μ_t que definem a política, bem como as redes alvo Q_{t1} e Q_{t2}, são atualizadas apenas a cada d iterações. Deste modo, as redes Q_{e1} e Q_{e2} são atualizadas com valores alvos de menor variância e, portanto, a convergência do algoritmo é aprimorada.

4 SIMULAÇÕES

Plataformas como a *OpenAI Baselines* oferecem implementações de diversos métodos de RL. Porém, para obter um melhor controle dos algoritmos utilizados neste projeto, estes foram desenvolvidos a partir de suas propostas originais e adaptados à tarefa de controle do sistema pêndulo invertido. Deste modo, diversas simulações foram desenvolvidas para analisar tanto os processos de treinamento quanto os controladores obtidos.

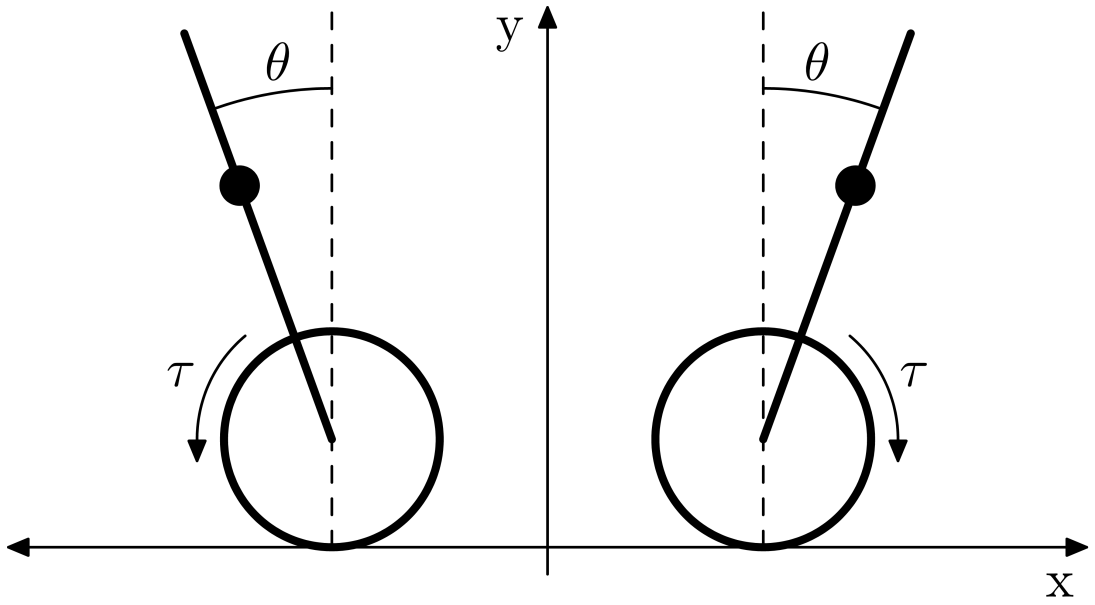
O presente trabalho foi desenvolvido e simulado em linguagem *Python* nos ambientes *Google Colab* e *Kaggle*. Para isso, as redes neurais, empregadas pelos métodos de RL aplicados, foram implementadas por meio da biblioteca *PyTorch*. Os códigos utilizados para implementação de ambos os agentes estão disponíveis no Apêndice A.

4.1 IMPLEMENTAÇÃO DOS CONTROLADORES

Métodos de RL não requerem conhecimento prévio sobre o sistema a ser controlado. Porém, informações a respeito do sistema pêndulo invertido foram utilizadas para aumentar a eficiência do aprendizado. Assim, o número de episódios e, consequentemente, o tempo de treinamento dos algoritmos desenvolvidos foram reduzidos de maneira significativa.

4.1.1 Simplificação por Simetria

Figura 17 – Exemplo de estados simétricos.



Fonte: Elaboração própria.

Os estados do sistema pêndulo invertido são simétricos em relação à posição $x = 0$, conforme exemplificado pelas duas situações da Figura 17. Diante de um estado s, em que $x < 0$ e sua ação ótima é dada por $a(s)$, o agente pode utilizar a ação $-a(-s)$ para obter o mesmo resultado ótimo. Portanto, o espaço de estados foi limitado para abranger apenas situações em que $x > 0$.

Deste modo, visto que os possíveis valores de entrada para as funções que devem ser aproximadas pelas redes neurais são reduzidos pela metade, esta tarefa de aproximação foi simplificada. Outra vantagem desta abordagem consiste na maior eficiência de dados utilizados para os treinamentos, pois os possíveis valores amostrados foram reduzidos. Assim, uma transição referente a um estado s pôde representar também o estado $-s$ e mais informações sobre o ambiente foram extraídas em cada iteração.

4.1.2 Redes Neurais

As redes neurais foram aplicadas com funções de ativação *ReLu* e otimizador *Adam*. Para a implementação do método TD3, foi necessário limitar os valores de ações fornecidas pelas redes neurais. Para isso, aplicou-se a função tangente hiperbólica na saída destas redes e, em seguida, os valores resultantes foram multiplicados pela tensão máxima aplicada no motor. Assim, as ações foram limitadas entre -12 V e 12 V. Para implementar o método *Double DQN*, este processo não foi necessário pois as ações geradas já são limitadas por valores discretos.

Algoritmos como o DDPG aplicam *batch normalization* nas redes neurais para que o agente apresente boa adaptação a diversos tipos de dados de entrada (LILLICRAP *et al.*, 2016). Nesta técnica, as redes neurais normalizam os dados processados para que estes apresentem médias e variâncias similares. Deste modo, a velocidade de treinamento pode ser aumentada.

Entretanto, para o caso deste projeto, a diferença de magnitude entre as variáveis de estado não foi considera significativa. A posição do robô x foi limitada entre -0,5 e 0,5 metros e a inclinação do pêndulo θ foi limitada em 20 graus, ou seja, aproximadamente 0,35 radianos. Verificou-se que, durante as simulações, as variáveis de velocidade $\frac{dx}{dt}$ e velocidade angular $\frac{d\theta}{dt}$ assumiam valores próximos às demais variáveis de estado. Portanto, não foi necessário utilizar *batch normalization*.

Por outro lado, as ações poderiam apresentar valores absolutos maiores em relação às variáveis de estado. Portanto, as ações fornecidas às redes neurais que calculam $Q(s, a)$ no método TD3 foram divididas por 50. Este número foi selecionado com base nos valores das variáveis de estado e das ações visualizadas durante simulações.

Para isso, visto que este número serve apenas para evitar valores extremos, não foi considerado necessário empregar maior rigor estatístico em sua escolha. Assim, esta adaptação requiriu pouco processamento computacional, não comprometeu o

funcionamento do algoritmo e contribuiu com o seu desempenho.

4.1.3 Função de Recompensa

Durante o treinamento, quando o agente se torna capaz de manter a posição e a inclinação do dispositivo próximas de zero, novas iterações passam a ocorrer em uma região restrita do espaço de estados. Portanto, as simulações foram realizadas por episódios limitados em cinco segundos. Após este período, inicia-se um novo episódio com estado inicial aleatório. Assim, estes estados iniciais contribuem com a exploração do espaço de estados.

Em outra situação, um episódio termina quando a posição ou a inclinação do robô passam de seus limites. Para que estes estados terminais fossem evitados, atribuiu-se a estes a pior recompensa possível, denominada R_T. Esta foi calculada pela recompensa em que tanto a inclinação quanto a posição do dispositivo encontram-se em seus valores limites. Assim, atingir tanto o limite da posição x quanto da inclinação θ representam a pior situação possível para o agente.

Além disso, considerou-se que o robô não possui capacidade de sair dos estados terminais. Deste modo, uma vez que estes estados são alcançados, todos os estados futuros retornam a mesma recompensa de estado terminal. Portanto, a recompensa acumulada, que considera as recompensas dos próximos estados ponderadas pelo fator de desconto γ, também apresenta o pior valor possível para os estados terminais.

Assim, a recompensa acumulada terminal, denominada G_T, pôde ser calculada conforme a equação (47), obtida a partir da equação (48). Deste modo, não houve a necessidade de calcular o valor Q do próximo estado para obter G_T. Desta maneira, os treinamentos das redes neurais ocorreram em menos episódios e com menor variância, pois suas atualizações não dependem de aproximações de valores Q, nestes estados terminais.

$$G_T = \frac{R_T}{1-\gamma} \tag{47}$$

$$G_T = \sum_{j=0}^{\infty} \gamma^j R_T \tag{48}$$

4.2 TREINAMENTOS

Com o objetivo de avaliar os métodos de RL aplicados neste trabalho, foram analisados, além dos controladores resultantes, os desempenhos dos treinamentos. Para isso, foi necessário realizar o ajuste de múltiplos hiperparâmetros presentes nestes métodos.

4.2.1 Escolha de Hiperparâmetros

Os hiperparâmetros utilizados são exibidos no Quadro 2 e no Quadro 3, para os métodos *Double DQN* e TD3, respectivamente. Para uma comparação correta entre estes métodos, os parâmetros que modificam a distribuição de recompensas conforme estados encontrados, como o fator de desconto γ, foram mantidos iguais entre os dois algoritmos.

Quadro 2 – Hiperparâmetros utilizados no método *Double DQN*

Tamanho das redes neurais	3.000 x 2.000 x 1.000 x 500 x 250
Tamanho de *batch* (N)	128
Tamanho de memória para *experience replay*	19.000
Fator de desconto (γ)	$0{,}99$
Taxa de aprendizado (α)	3.10^{-6}
Episódios por atualização de redes alvo (C)	400
Número de possíveis ações	9
ε inicial	100%
ε final	1%
Decaimento de ε por iteração	$0{,}1\%$

Fonte: Elaboração própria.

Quadro 3 – Hiperparâmetros utilizados no método *TD3*

Tamanho das redes neurais	2.000 x 1.000 x 500 x 250
Tamanho de *batch* (N)	128
Tamanho de memória para *experiênce replay*	Não limitado
Fator de desconto (γ)	$0{,}99$
Taxa de aprendizado (α)	5.10^{-5}
Episódios por atualização da rede atuadora (d)	2
Desvio padrão do ruído adicionado em ações aplicadas pelas redes atuadoras	10 mV
Desvio padrão do ruído adicionado em ações utilizadas pelas redes críticas	120 mV
Número de ações iniciais aleatórias (D)	3.000
Fator de *Polyak Averaging* (T)	5.10^{-3}

Fonte: Elaboração própria.

Em aplicações práticas, o ajuste de hiperparâmetros pode ser essencial para se obter boa performance em treinamentos e controladores. Porém, apesar de haver diversos métodos que realizam a seleção automática de parâmetros ótimos (YANG; SHAMI, 2020), estes não foram regulados de forma automática. Portanto, espera-se que um ajuste fino destes valores possa produzir resultados mais eficazes.

A escolha deste tipo de ajuste foi justificada pelo tempo necessário para explorar diferentes configurações de hiperparâmetros. Ambos os algoritmos aplicados possuem

dezenas destes parâmetros. Além disso, cada treinamento durou entre uma e quatro horas para o método *Double DQN* e entre quatro e oito horas para o método TD3. Deste modo, visto que o número de possíveis combinações de hiperparâmetros varia de maneira exponencial em função do número destes parâmetros e seus valores testados, uma exploração mais extensa destas combinações se tornou custosa demais para este trabalho.

Ademais, algoritmos de otimização de hiperparâmetros requerem métricas de avaliação que devem ser maximizadas nestes processos. Porém, diversas métricas de avaliação podem ser aplicadas e nenhuma destas representa o desempenho do treinamento ou do controlador de maneira ideal.

Portanto, métodos de RL são tipicamente avaliados por gráficos e tabelas de recompensa acumulada (HENDERSON *et al.*, 2018). Assim, além de resultados numéricos, como por exemplo recompensa acumulada máxima, variâncias e tempo de treinamento, analisou-se a evolução destes valores ao longo de cada treinamento.

4.2.2 Análise de Custo Acumulado e Duração de Episódios

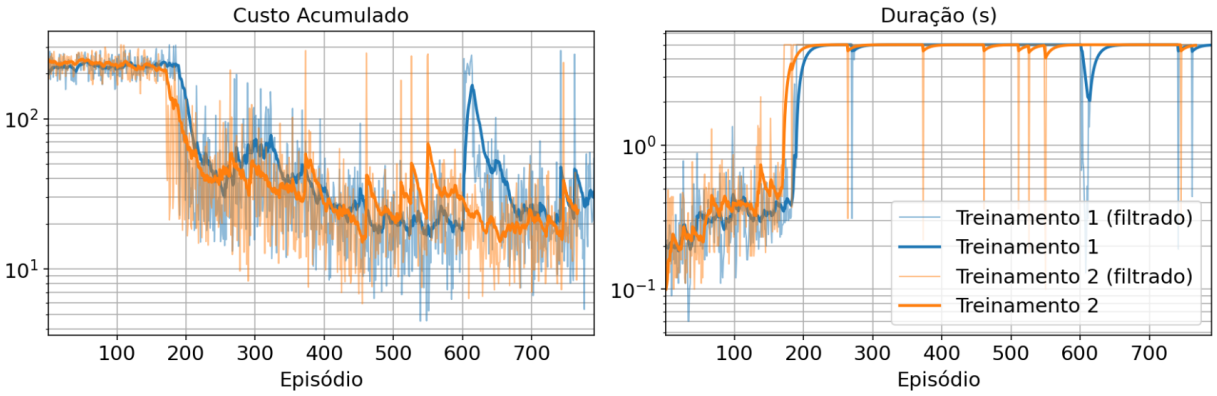
Os treinamentos foram avaliados, principalmente, a partir da evolução do sinal de custo acumulado por episódio, calculado pelo valor negativo da recompensa acumulada por episódio. Além disso, avaliou-se a duração de cada episódio, que indica por quanto tempo o agente foi capaz de manter o pêndulo entre seus limites de inclinação e posição.

Deste modo, o controlador deve aprender a minimizar o custo acumulado e manter a duração dos episódios no valor máximo de cinco segundos. A performance dos treinamentos foi caracterizada, também, pelo número de episódios, variância dos sinais analisados e variações entre treinamentos.

Exemplos de resultados de treinamentos com ambos os métodos aplicados são apresentados nas Figuras 18 e 19. Devido a alterações na política aprendida durante os treinamentos e diferenças entre estados iniciais, os sinais analisados variaram entre episódios de maneira significativa. Portanto, para uma melhor compreensão do comportamento destes sinais, estes são exibidos em conjunto com o resultado da aplicação de um filtro passa-baixa do tipo *Polyak Averaging*.

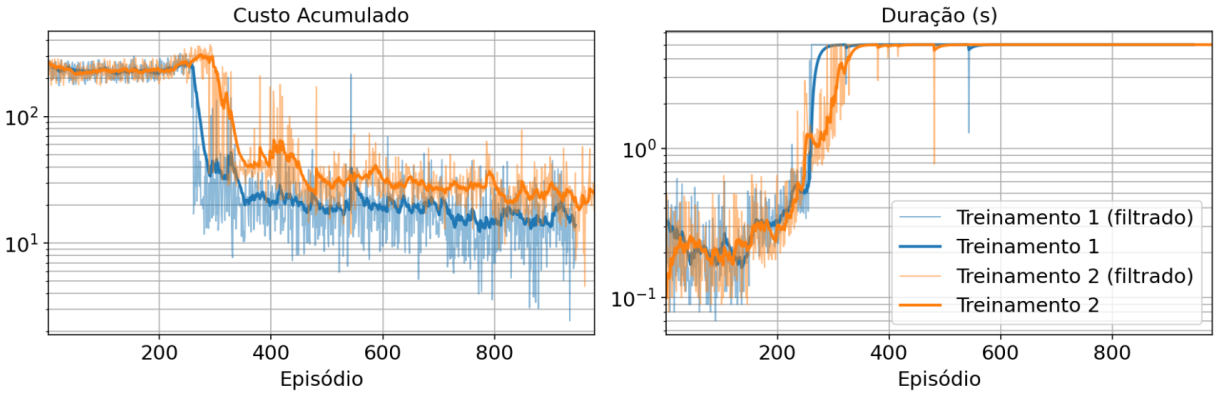
Nota-se que, mesmo após os controladores aprenderem a manter o pêndulo invertido equilibrado por múltiplos episódios, estes podem falhar em episódios seguintes. Isto indica que, apesar de não comprometer a convergência dos algoritmos, algumas atualizações de pesos das redes neurais podem prejudicar o desempenho dos controladores. Este efeito ocorreu com maior frequência no algoritmo *Double DQN* e foi atenuado, principalmente, pelo aumento do tamanho das redes neurais.

Figura 18 – Exemplo de treinamentos com o método *Double DQN*.



Fonte: Elaboração própria.

Figura 19 – Exemplo de treinamentos com o método TD3.



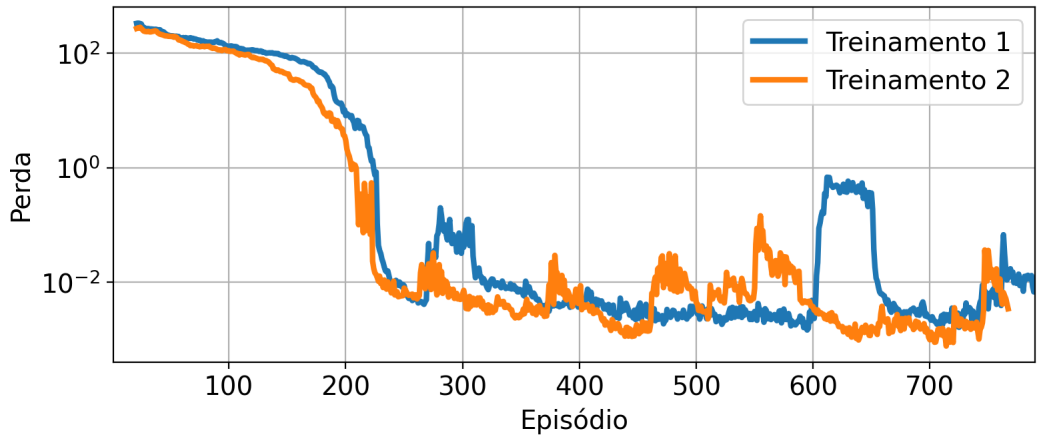
Fonte: Elaboração própria.

4.2.3 Função Perda

Outro sinal monitorado durante os treinamentos consiste na perda utilizada pelas redes neurais aproximadoras de valores Q. Exemplos destes sinais obtidos a partir de treinamentos com o método *Double DQN*, com aplicação do filtro passa-baixa, são exibidos na Figura 20.

Deste modo, a perda indica o erro de aproximação dos valores Q encontrados ao longo do treinamento. Entretanto, um aumento na perda não significa, necessariamente, uma piora da política em aprendizagem. Isto ocorre pois, ao modificar esta política, novas regiões do espaço de estados passam a ser exploradas. Assim, as redes aproximadoras de valores Q apresentam dificuldades em avaliar estes novos dados.

Figura 20 – Exemplo de perdas utilizadas pelas redes neurais aproximadoras do valor Q durante treinamentos com o método *Double DQN*.

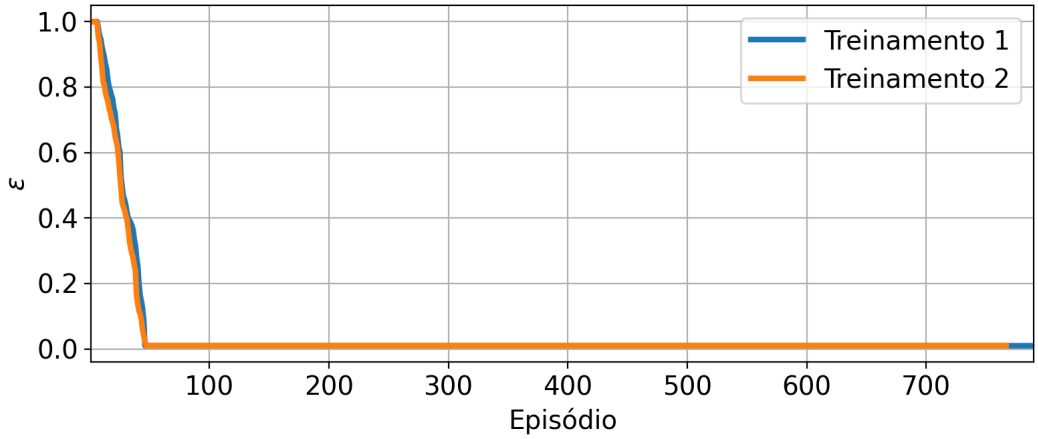


Fonte: Elaboração própria.

4.2.4 Exploração e Avaliação

Conforme justificado na seção 3.2, o algoritmo *Double DQN* aplica ações aleatórias com probabilidade ε durante o treinamento. Este parâmetro foi reduzido até o valor mínimo de 1%, conforme exibido nos exemplos da Figura 21.

Figura 21 – Exemplos de variação do parâmetro ε durante treinamentos.



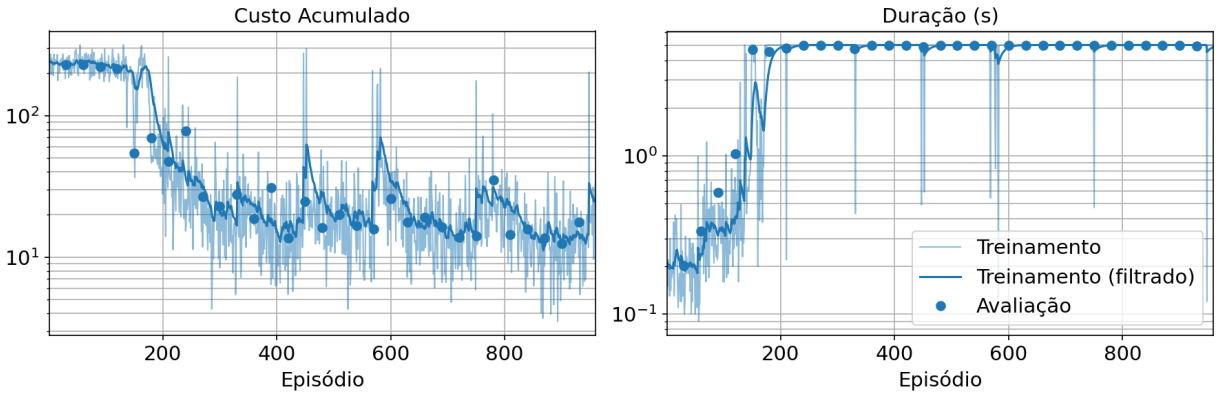
Fonte: Elaboração própria.

Nota-se que o efeito de ações aleatórias foi predominante apenas nos episódios iniciais. Porém, esta interferência nas ações da política em aprendizagem ocorreu durante todo o treinamento. Deste modo, após finalizar o aprendizado, espera-se que o controlador produza resultados melhores que os vistos durante este processo. De maneira semelhante, o algoritmo TD3 adiciona ruído em suas ações apenas durante o treinamento.

Portanto, para medir o custo acumulado obtido pela política ainda em processo de treinamento, sem a interferência de ações aleatórias, aplicou-se simulações de avaliação. Neste caso, os treinamentos foram interrompidos temporariamente, a cada

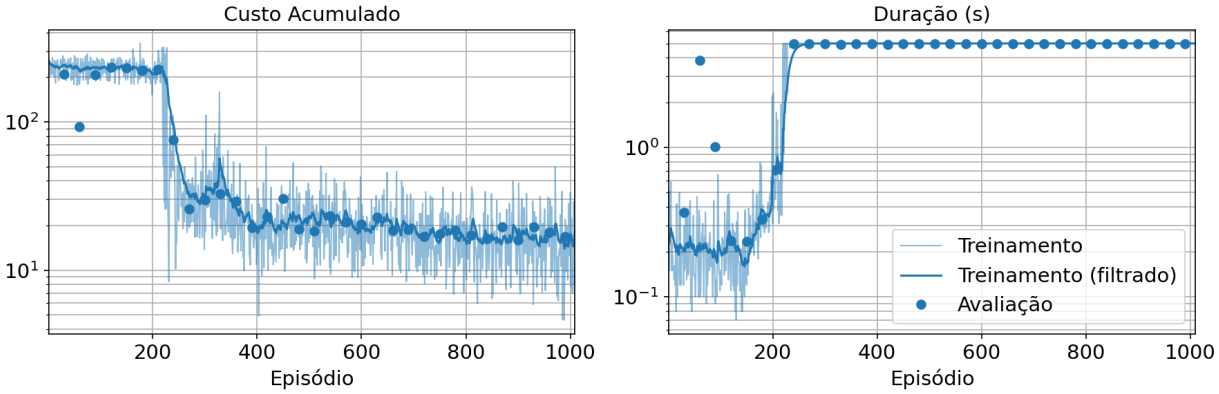
trinta episódios, para testar as políticas em desenvolvimento. Nestes testes, as políticas foram avaliadas pelo custo acumulado médio de cem episódios. Os resultados de ambos os métodos aplicados são apresentados nos gráficos das Figuras 22 e 23.

Figura 22 – Exemplo de treinamento com avaliações do agente *Double DQN*.



Fonte: Elaboração própria.

Figura 23 – Exemplo de treinamento com avaliações do agente TD3.



Fonte: Elaboração própria.

Nota-se que os sinais obtidos durante os treinamentos podem representar o resultado de avaliações de maneira satisfatória. Portanto, visto que o processo de avaliação requer um custo computacional significativo, este não foi utilizado no ajuste de hiperparâmetros.

4.2.5 Desempenho dos Treinamentos

Em razão de variações de resultados entre treinamentos e a falta de métricas de avaliação padrões, múltiplos treinamentos devem ser executados para assegurar a reprodução e a interpretação correta destes resultados (HENDERSON *et al.*, 2018).

Deste modo, verificou-se o número de episódios necessários para que os agentes treinados aprendessem, no decorrer dos treinamentos, a manter o pêndulo invertido

equilibrado durante todo um episódio de cinco segundos. O resultado de medidas com quarenta treinamentos de cada método aplicado é exibido na Tabela 2.

Tabela 2 – Episódios necessários para manter o pêndulo invertido equilibrado por cinco segundos.

Método	**Média**	**Desvio padrão**
Double DQN	193,3	31,7
TD3	208,9	46,1

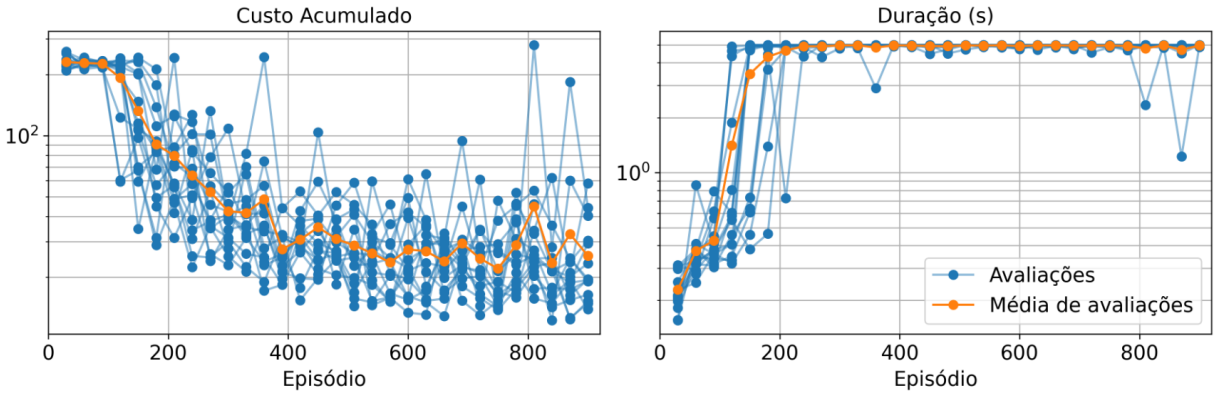
Fonte: Elaboração própria.

Para uma comparação, cita-se um projeto similar, realizado por Bates (2021), que controlou um sistema do tipo pêndulo invertido com o método de RL *Policy Gradient*. Neste caso, relatou-se que foi obtido sucesso ao equilibrar o pêndulo com em média 355 episódios, porém, os valores medidos variaram entre 54 e 2000 episódios.

Assim, pôde-se concluir que ambos os métodos aplicados no presente projeto foram capazes de resolver a tarefa de controle proposta de maneira satisfatória. Porém, buscou-se agentes que, além de manter o equilíbrio do robô, fossem capazes de minimizar o custo acumulado. Deste modo, obteve-se controladores mais eficazes e robustos.

Apresenta-se nas figuras 24 e 25 o resultado de múltiplas avaliações de políticas durante treinamentos e seus valores médios. Para esta medida, foram realizados quinze treinamentos com cada método aplicado.

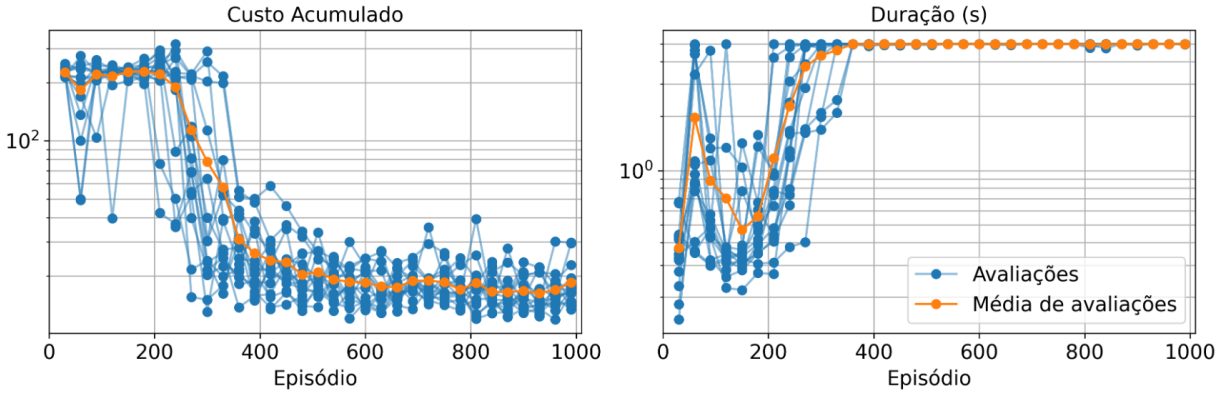
Figura 24 – Média de avaliações periódicas de controladores em processo de treinamento com o método *Double DQN*.



Fonte: Elaboração própria.

Nota-se que as melhores políticas não foram obtidas necessariamente no fim dos treinamentos, pois alguns episódios degradaram o desempenho das políticas em aprendizagem. Portanto, em cada treinamento, os parâmetros dos controladores foram salvos sempre que uma avaliação da política fornecia um resultado melhor.

Figura 25 – Média de avaliações periódicas de controladores em processo de treinamento com o método TD3.



Fonte: Elaboração própria.

Em seguida, verificou-se o custo acumulado obtido pelas políticas resultantes dos treinamentos e o número de episódios em que estas foram encontradas. A média e o desvio padrão destes valores são apresentados na Tabela 3. Para isso, realizou-se vinte treinamentos com cada algoritmo aplicado. Os treinamentos de ambos os métodos foram executados por 4.10^5 iterações de 10 ms, com avaliações de política a cada 5 episódios.

Tabela 3 – Análise de resultados de treinamentos.

Método	**Custo acumulado**		**Episódio com melhor política**	
	Média	**Desvio padrão**	**Média**	**Desvio padrão**
Double DQN	13,00	0,75	873,0	110,6
TD3	11,65	0,69	783,8	177,2

Fonte: Elaboração própria.

Em geral, o método TD3 apresentou melhores resultados. Em alguns treinamentos, este foi capaz de equilibrar o pêndulo invertido com apenas trinta episódios. Porém, devido a variações entre treinamentos, mais episódios podem ser necessários para produzir uma política satisfatória. Assim, a depender de requisitos de projetos e da viabilidade de implementação dos treinamentos, o número de episódios e treinamentos executados podem ser ajustados para aprimorar a eficácia da política resultante.

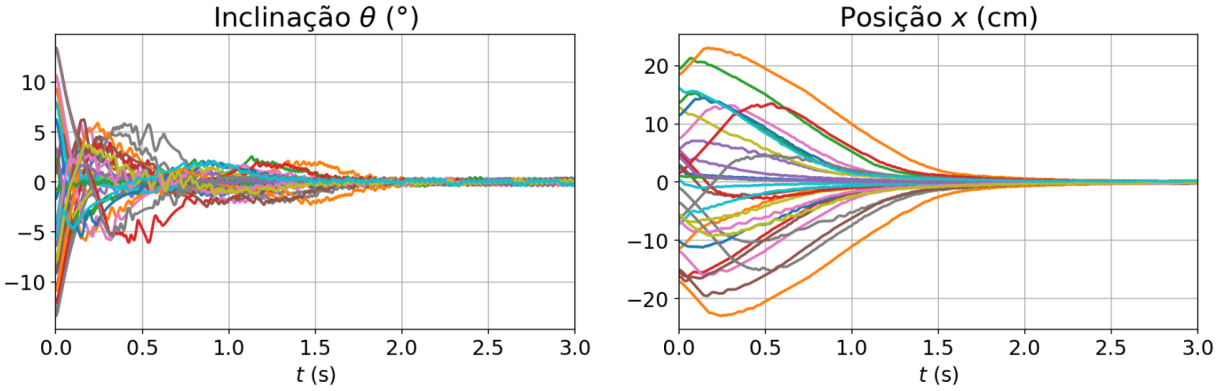
4.3 CONTROLADORES RESULTANTES

Com o objetivo de avaliar as políticas obtidas, realizou-se simulações com diversos estados iniciais. Além disso, foi aplicado um sinal de entrada para controlar a posição do robô.

4.3.1 Análise de Políticas

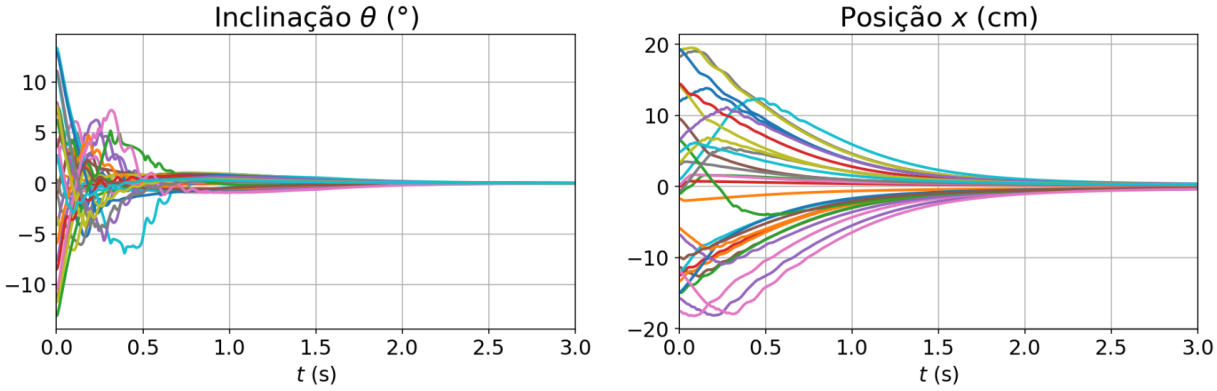
Os resultados de diversas simulações, com estados iniciais similares aos utilizados durante os treinamentos, são apresentados nas figuras 26 e 27, para os dois métodos de RL aplicados.

Figura 26 – Simulações com o método *Double DQN* após aprendizado.



Fonte: Elaboração própria.

Figura 27 – Simulações com o método TD3 após aprendizado.



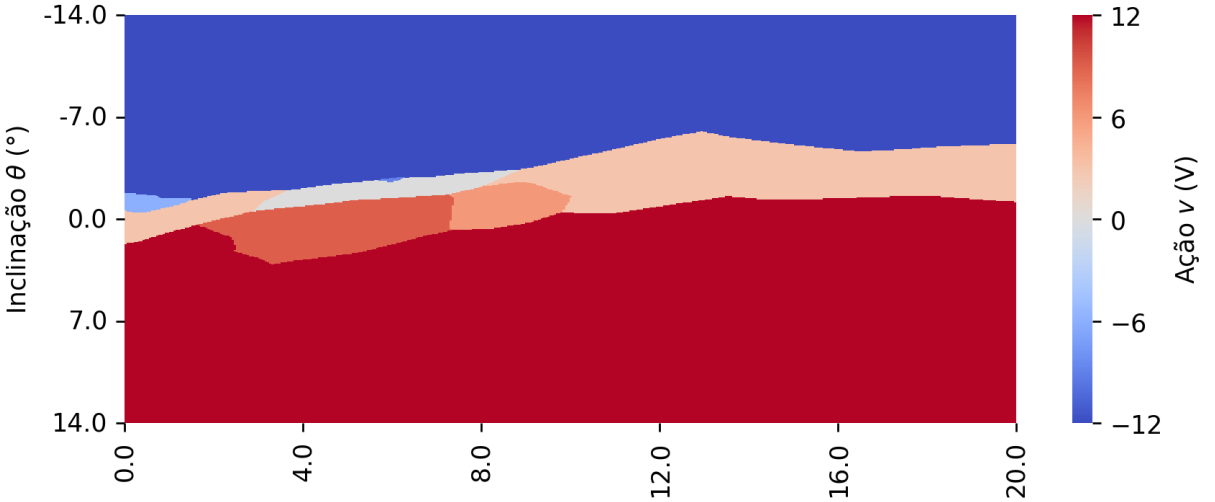
Fonte: Elaboração própria.

Nota-se que a tarefa de equilíbrio foi resolvida com sucesso por ambos os métodos aplicados. Entretanto, considerando as variações dos sinais avaliados e o tempo necessário para ajustar a inclinação θ e a posição x do robô, o método TD3 apresentou melhores resultados.

Em alguns casos, com algoritmo TD3 e a ausência de perturbações, tanto a inclinação θ quanto a posição x convergiram para zero. Por outro lado, estes mesmos sinais, quando controlados pela política obtida pelo método *Double DQN*, apresentaram variações durante todo o episódio. Mesmo em episódios mais longos, as amplitudes destas variações permaneceram em, no mínimo, 0,4° e 3 mm.

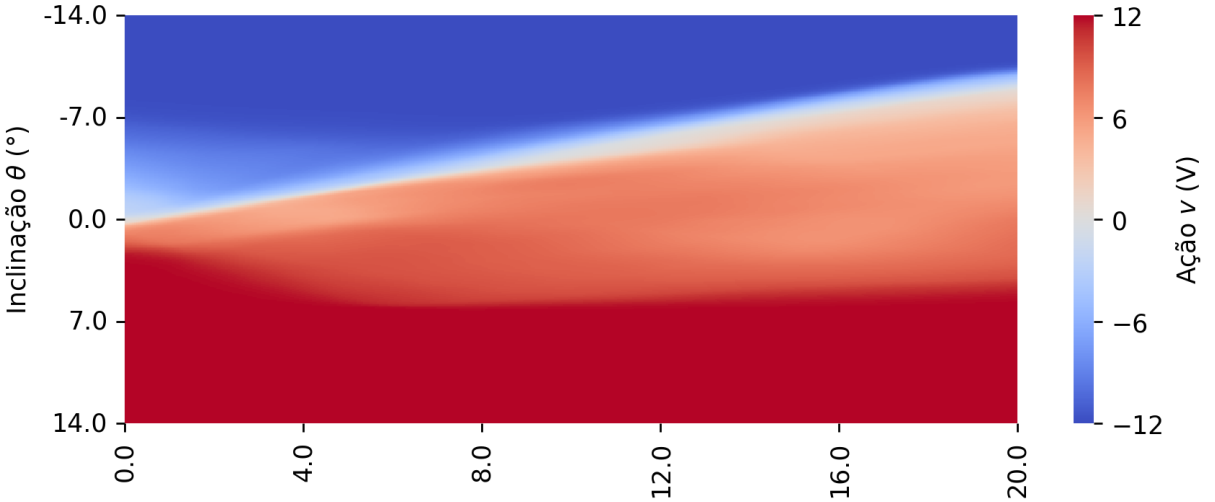
Devido à alta dimensionalidade do espaço de estados, não foi possível visualizar a escolha de ações em função de todos os estados. Portanto, as ações selecionadas pelos controladores foram plotadas em função da inclinação θ e da posição x, com as demais variáveis de estado nulas, nas figuras 28 e 29. A região em que $x < 0$ não é exibida devido à simetria do espaço de estados.

Figura 28 – Ações escolhidas pelo agente *Double DQN*, após treinamento, em função de variáveis de estado.



Fonte: Elaboração própria.

Figura 29 – Ações escolhidas pelo agente TD3, após treinamento, em função de variáveis de estado.



Fonte: Elaboração própria.

Nota-se que as ações com valor absoluto máximo, dadas por -12 V e 12 V, foram predominantes. Deste modo, as ações de valores intermediários foram utilizadas para

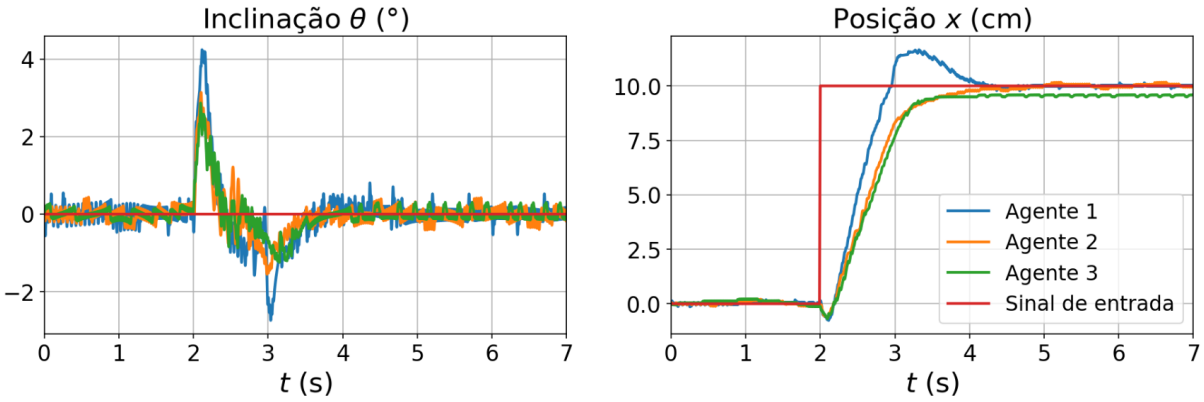
ajustes mais finos da inclinação e da posição. Isto indica que uma faixa maior de possíveis valores de tensão poderia resultar em políticas melhores. Para isso, alterações no motor ou demais configurações do projeto podem ser investigadas.

Percebe-se que, em razão da maior liberdade na escolha de ações, o método TD3 possui potencial para produzir políticas com melhor desempenho em relação ao método *Double DQN*. Além disso, o sistema pêndulo invertido não possui descontinuidades em suas equações de estados. Portanto, estima-se que a política ótima também não apresente descontinuidades.

4.3.2 Aplicação de Sinal de Entrada

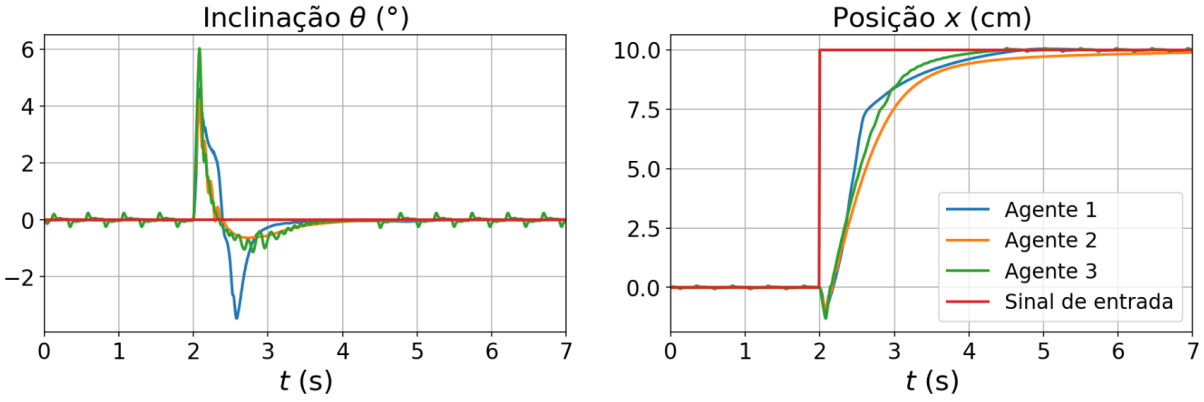
Os controladores deste trabalho foram treinados para manter a inclinação θ e a posição x nulas. Porém, ao variar a referência de medida da posição x, pôde-se controlar esta variável de estado por um sinal de entrada. Para isso, aplicou-se um sinal de entrada degrau e verificou-se o comportamento do sistema com três controladores distintos, para cada método, conforme as Figuras 30 e 31.

Figura 30 – Simulações com sinal de entrada degrau e método *Double DQN*.



Fonte: Elaboração própria.

Figura 31 – Simulações com sinal de entrada degrau e método TD3.



Fonte: Elaboração própria.

Ao analisar os resultados do algoritmo *Double DQN*, percebeu-se que o agente 1 apresentou uma resposta com *overshooting*, diferente dos demais agentes. No caso do método TD3, observou-se que apenas os agentes 1 e 2 foram capazes de reduzir os erros de inclinação e posição à exatamente zero.

Além disso, percebe-se que, para aumentar a variável de estado x, esta deve primeiro ser reduzida em uma manobra de correção da inclinação θ.

Nota-se que ambos os métodos apresentaram controladores com comportamentos distintos, porém o método *Double DQN* apresentou mais variações entre seus controladores resultantes. Duas justificativas para estas variações são a diferença entre inicializações de parâmetros das redes neurais e a diferença entre estados iniciais de episódios de treinamentos.

4.4 CONSIDERAÇÕES PARA APLICAÇÕES PRÁTICAS

Os agentes treinados neste trabalho possuem acesso a todas as variáveis de estado que representam o sistema controlado. Porém, em situações práticas, estas informações devem ser adquiridas através de sensores que apresentam erros inerentes em suas medidas. Além disso, as ações aplicadas em forma de tensão no motor também estão suscetíveis a erros. Portanto, esta seção trata destas dificuldades práticas.

4.4.1 Sensores

A posição e a velocidade de robôs do tipo pêndulo invertido são tipicamente medidas por sensores *encoders* (KLÖPPELT; MEYER, 2018). Nota-se que, se estes forem acoplados diretamente ao eixo do motor, erros de medida devido à zona morta da caixa de redução mecânica podem aumentar a incerteza deste sinal de maneira significativa (GRASSER *et al.*, 2002).

Por outro lado, os dados de inclinação e velocidade angular são tipicamente obtidos por unidades de medição inercial (IMU, do inglês *Inertial Measurement Unit*), conforme demonstrado por Mellatshahi *et al.* (2021). Esta é composta pelo sensor giroscópio, para medir a velocidade angular, e pelo sensor acelerômetro, para medir a aceleração percebida pelo dispositivo.

Deste modo, a inclinação pode ser obtida através da integração do sinal fornecido pelo giroscópio. Porém, esta medida contém erros de *drift* que aumentam ao longo do tempo, pois as medidas de velocidade angular podem incluir erros com média não nula.

De modo alternativo, o sensor acelerômetro tem a capacidade de medir a inclinação do dispositivo ao identificar a direção da aceleração gravitacional. Porém, demais acelerações impostas ao robô dificultam a separação desta componente de aceleração.

Portanto, a incerteza desta medida varia de acordo com movimentos do dispositivo.

Visto que os sensores giroscópio e acelerômetro possuem erros de naturezas distintas, tipicamente se aplicam técnicas, como o filtro de Kalman, para se obter uma medida de posição angular mais precisa com base em ambos estes sensores (BEARD; MCLAIN, 2012).

Um modelo detalhado destes sensores foge do escopo deste trabalho. Portanto, os erros de medidas das variáveis de estados foram modelados pela adição de ruídos com distribuição aleatória gaussiana.

4.4.2 Simulações com Perturbações

Além de perturbações causadas por erros de medidas, os erros das ações executadas pelos agentes também foram modelados. Para isso, o sinal de tensão no motor v também teve um ruído adicionado.

Demais perturbações, como forças externas aplicadas ao robô, foram desconsideradas. Assim, de acordo com os sensores analisados, estimou-se ruídos gaussianos com médias nulas e os desvios padrões exibidos no Quadro 4.

Quadro 4 – Desvios padrões dos ruídos gaussianos, de acordo com cada variável de estado, aplicados no modelo de perturbações

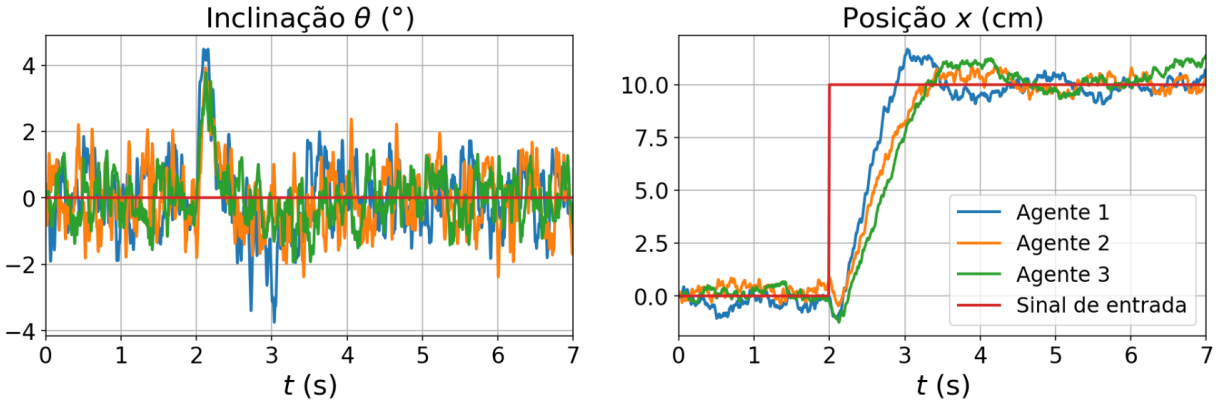
Posição ($\mathbf{x}$)	10 mm
Velocidade ($\frac{\mathbf{dx}}{\mathbf{dt}}$)	10 mm/s
Posição Angular ($\boldsymbol{\theta}$)	$1{,}0$ °
Velocidade Angular ($\frac{\mathbf{d}\boldsymbol{\theta}}{\mathbf{dt}}$)	$0{,}1$ °/s
Tensão ($\mathbf{v}$)	$0{,}1$ V

Fonte: Elaboração própria.

Deste modo, as simulações com sinal de entrada degrau apresentadas na seção anterior, com os mesmos agentes, foram executadas com o modelo de perturbações. Os resultados são exibidos nas figuras 32 e 33 para os métodos *Double DQN* e TD3, respectivamente.

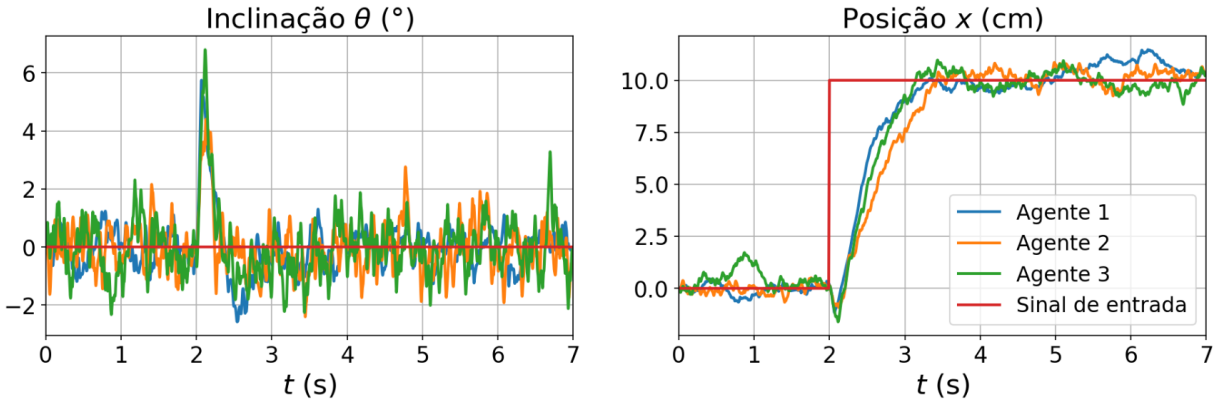
Com este nível perturbação, ambos os métodos de RL foram capazes de manter o pêndulo equilibrado. Porém, percebe-se uma variância maior nos sinais controlados. Em especial, o método TD3 passou a não ser mais capaz de manter os erros destes sinais nulos.

Figura 32 – Simulações com sinal de entrada degrau e método *Double DQN* com ruído.



Fonte: Elaboração própria.

Figura 33 – Simulações com sinal de entrada degrau e método TD3 com ruído.



Fonte: Elaboração própria.

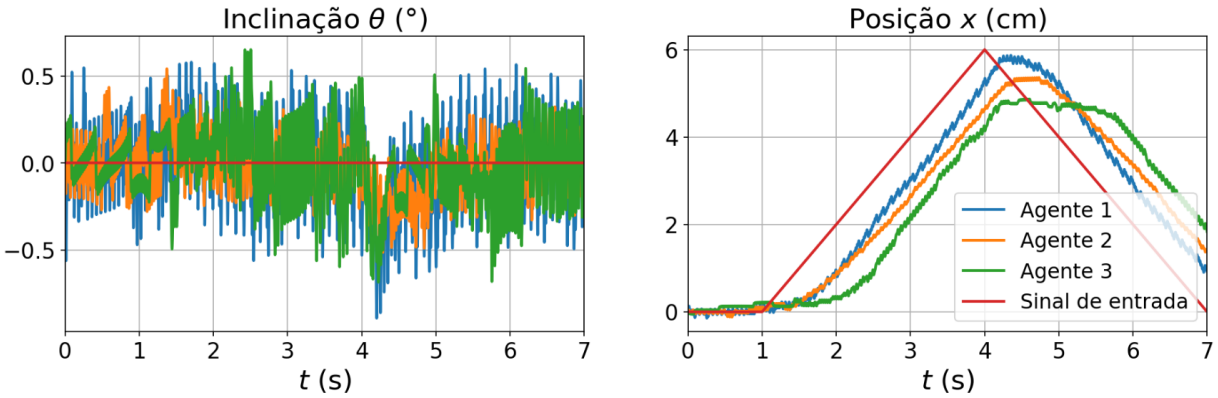
4.4.3 Ajuste de Recompensa

Em aplicações práticas, pode ser necessário variar a posição x do robô de maneira constante. Portanto, foram realizadas simulações com uma onda triangular como sinal de entrada, conforme exibido nas Figuras 34 e 35. Desta vez, com o objetivo de enfatizar os efeitos de variação do sinal de entrada, não foi aplicado ruído.

Nota-se que os controladores foram capazes de seguir o sinal de entrada, porém, em relação às simulações com sinal degrau, houve um aumento significativo nos erros de controle da posição x.

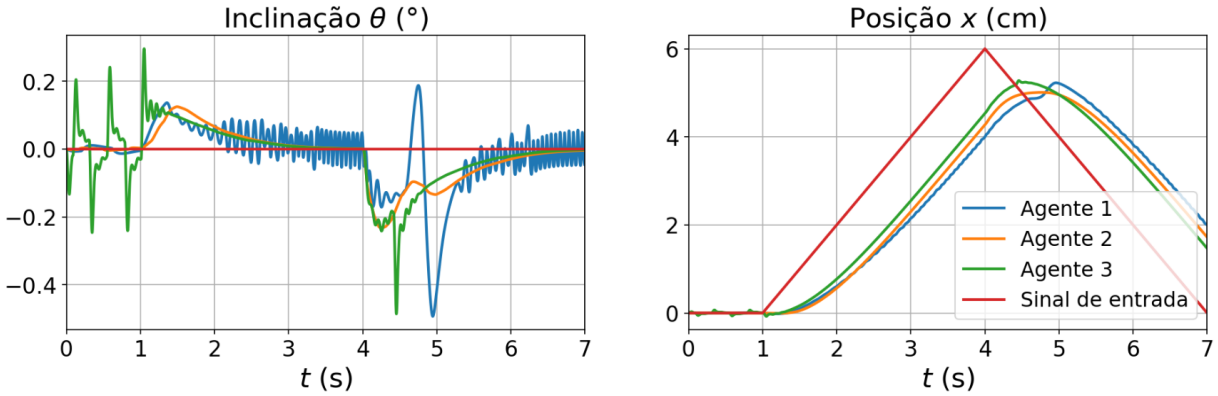
A transição do sinal degrau aplicado anteriormente pode ser interpretada como uma tarefa similar às apresentadas durante os episódios de treinamento. No caso dos treinamentos, as transições dos sinais foram definidas pelos valores iniciais das variáveis de estados. Porém, um sinal do tipo rampa não foi observado pelos agentes durante os treinamentos. Deste modo, justifica-se uma maior dificuldade para os controladores se adaptarem a este tipo de sinal.

Figura 34 – Simulações com sinal de entrada rampa e método *Double DQN*.



Fonte: Elaboração própria.

Figura 35 – Simulações com sinal de entrada rampa e método TD3.



Fonte: Elaboração própria.

Uma possível solução para este problema consiste na aplicação de sinais com formas variadas durante o processo de treinamento. Porém, como uma alternativa a esta solução, a função recompensa foi modificada. Deste modo, os fatores da equação (30), apresentada no seção 3.1, foram modificados para tornar o controle da variável de posição x mais importante que o controle da variável de inclinação θ. Assim, o fator λ_x foi triplicado e obteve-se os resultados exibidos nas Figuras 36 e 37.

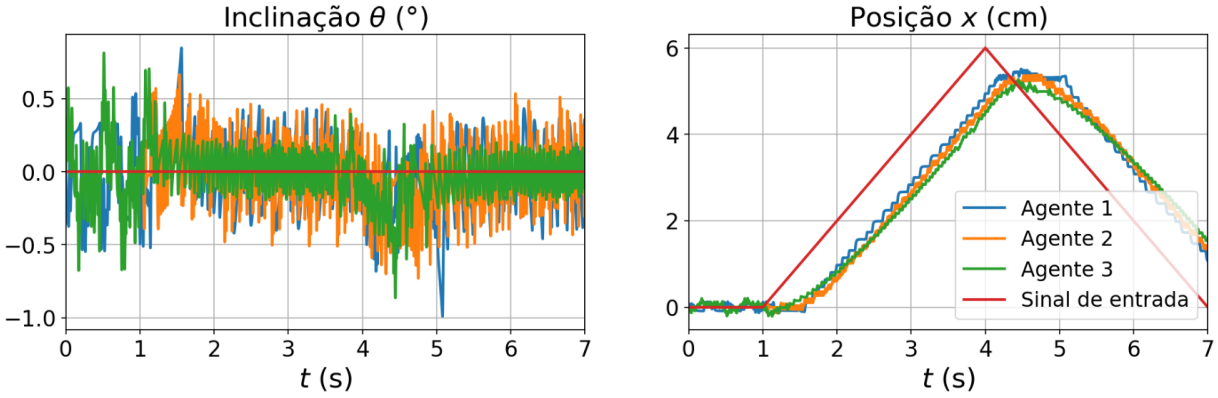
Nota-se que, desta maneira, um melhor controle sobre a posição x pôde ser obtido a custo de uma degradação no controle da inclinação θ. Entretanto, a mudança no comportamento dos agentes obtidos com o método *Double DQN* foi mais sutil.

Outra possibilidade de manipulação da política resultante consiste na variação do fator de desconto γ. Assim, a posição x pode ser corrigida de maneira mais veloz, em troca de uma maior inclinação temporária do pêndulo durante esta manobra.

Além disso, diferentes tipos de funções podem ser utilizadas para modelar o sinal de recompensa. Como por exemplo, pode-se incluir a velocidade do robô, consumo de energia do motor ou termos exponenciais em função das variáveis de estados.

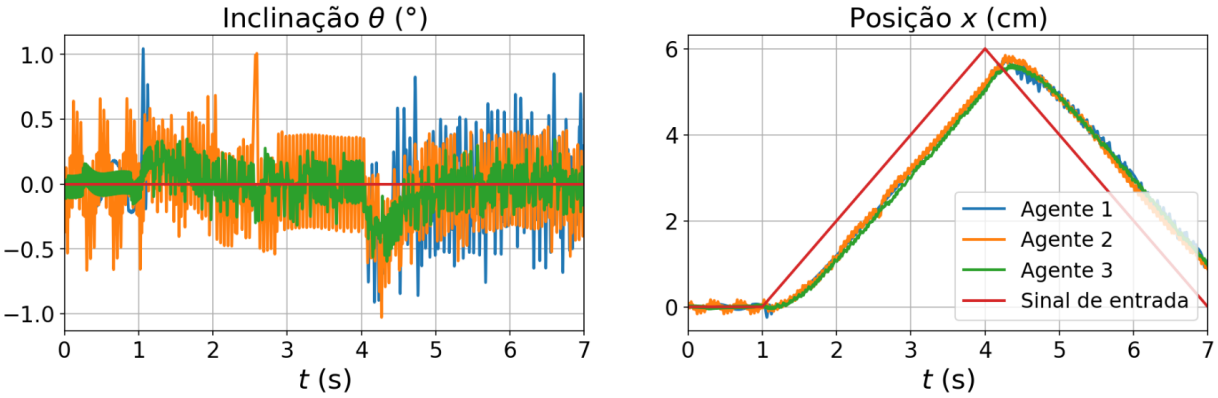
Deste modo, a liberdade de personalização do objetivo do controlador de acordo com o projeto consiste em vantagem de métodos de RL em relação a métodos de controle clássicos.

Figura 36 – Simulações com sinal de entrada rampa e método *Double DQN*, após ajuste do parâmetro λ_x na função recompensa.



Fonte: Elaboração própria.

Figura 37 – Simulações com sinal de entrada rampa e método TD3, após ajuste do parâmetro λ_x na função recompensa.



Fonte: Elaboração própria.

5 CONCLUSÃO

O presente trabalho tem o objetivo de, a partir de métodos de inteligência artificial, controlar o equilíbrio de um robô com estabilidade em duas rodas. Para tanto, foram aplicados os controladores *Double Deep Q-Network* e *Twin Delayed Deep Deterministic Policy Gradient*.

O robô objeto deste estudo constitui um sistema do tipo pêndulo invertido, presente em diversas aplicações práticas, como robôs utilizados para locomoção humana ou transporte de objetos. Deste modo, espera-se que as informações produzidas, tanto sobre o modelo do sistema quanto sobre os controladores, possam ser empregadas em múltiplos sistemas similares.

Os controladores foram aplicados de maneira virtual, portanto, com a finalidade de se obter simulações mais próximas da realidade, o robô foi modelado a partir de medidas de parâmetros de um motor real. Assim, além de analisar os controladores desenvolvidos, este projeto considerou diversos efeitos de aplicações práticas.

Ambos os algoritmos aplicados foram capazes de completar a tarefa de controle com sucesso. Porém, o método *Twin Delayed Deep Deterministic Policy Gradient* apresentou desempenhos melhores. Esta diferença foi justificada, principalmente, devido à limitação de ações em domínio discreto imposta pelo algoritmo *Double Deep Q-Network*.

Tipicamente, projetos similares com sistemas do tipo pêndulo invertido controlaram apenas a inclinação do pêndulo, enquanto a localização do dispositivo foi mantida em um espaço limitado mas não controlada. Porém, a tarefa de controle do presente trabalho foi definida pelo domínio de tanto a inclinação do robô quanto a localização do centro de suas rodas, medida em relação ao plano em que este se desloca. Além disso, aplicou-se um sinal de entrada para controlar a posição do robô.

O agente *Twin Delayed Deep Deterministic Policy Gradient* constitui um método recente de aprendizado por reforço. Desde modo, durante a revisão bibliográfica, não foram encontradas aplicações deste algoritmo com o objetivo formulado da mesma forma empregada pelo presente trabalho.

Foram encontradas diversas aplicações do método *Deep Q-Network* em sistemas do tipo pêndulo invertido, inclusive com robôs reais. Porém, os trabalhos que utilizaram a variação *Double Deep Q-Network* não consideraram o controle da posição do robô.

Os algoritmos aplicados neste trabalho pertencem à área de aprendizado por reforço, em que os controladores são treinados através de tentativas e erros para resolver a tarefa de controle. Deste modo, esta solução constitui uma alternativa a métodos de controle clássicos. Assim, fatores como não linearidade e instabilidade do sistema não apresentaram dificuldades, de modo significativo, no desenvolvimento dos

controladores.

Em contrapartida, outras dificuldades foram encontradas no desenvolvimento deste projeto. O objetivo do controlador foi definido com maior liberdade ao utilizar métodos de aprendizado por reforço, em relação a métodos de controle clássicos. Porém, os impactos desta definição não são evidentes. Portanto, múltiplos experimentos foram necessários para avaliar o comportamento dos controladores.

Além disso, os algoritmos aplicados neste trabalho possuem hiperparâmetros que demandaram diversos ajustes, por simulações, para garantir o funcionamento destes métodos. Portanto, visto que cada treinamento dos controladores durou cerca de seis horas, o desenvolvimento deste projeto foi limitado pelo tempo de processamento computacional.

Em razão de erros inerentes dos modelos virtuais, propõe-se para trabalhos futuros a aplicação prática dos controladores desenvolvidos. Neste caso, a elaboração de treinamentos pode ser mais complexa e custosa. Portanto, o modelo projetado no presente trabalho pode ser utilizado para realizar treinamentos virtuais de maneira completa ou como inicialização para serem finalizados com o robô real.

Verificou-se que a indutância do motor possui pouca influência no comportamento do sistema e, por consequência, os modelos foram simplificados. Porém, efeitos não modelados, como a zona morta de movimento em reduções mecânicas, podem afetar o desempenho dos controladores de maneira significativa. Em vista disso, sugere-se o estudo de modelos mais realistas para robôs do tipo pêndulo invertido.

Este trabalho considerou os efeitos de ruídos nos sinais de sensores e na tensão aplicada no motor, porém, apenas durante a avaliação dos controladores. Portanto, recomenda-se o estudo dos efeitos de perturbações incluídas nos processos de treinamento dos controladores.

Além disso, visto que alguns hiperparâmetros dos controladores podem afetar o comportamento do sistema de maneira significativa, sugere-se a avaliação dos impactos de alterações nos hiperparâmetros. Em especial, pode-se verificar a variação do tempo de resposta dos controladores ao modificar o fator de desconto γ.

A inteligência artificial constitui uma área que abrange diversos métodos de controle e segue em constante crescimento. Portanto, para obter melhores desempenhos com o sistema pêndulo invertido, recomenda-se o estudo de outros controladores, técnicas de aprendizagem e algoritmos de ajuste automatizado de hiperparâmetros.

REFERÊNCIAS

ABIODUN, Oludare Isaac; JANTAN, Aman; OMOLARA, Abiodun Esther; DADA, Kemi Victoria; MOHAMED, Nachaat AbdElatif; ARSHAD, Humaira. State-of-the-art in artificial neural network applications: A survey. **Heliyon**, v. 4, n. 11, 2018.

ADAM, Sander; LUCIAN, Busoniu; ROBERT, Babuska. Experience Replay for Real-Time Reinforcement Learning Control. **IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)**, v. 42, n. 2, p. 201–212, 2011.

AUNG, Wai Phyo. Analysis on Modeling and Simulink of DC Motor and its Driving System Used for Wheeled Mobile Robot. **World Academy of Science, Engineering and Technology**, v. 1, n. 8, p. 1149–1156, 2007.

AWODA, Murtadha; ALI, Ramzy. Parameter Estimation of a Permanent Magnetic DC Motor. **Iraqi Journal for Electrical and Electronic Engineering**, v. 15, n. 1, p. 28–36, 2019.

BATES, Dylan. A Hybrid Approach for Reinforcement Learning Using Virtual Policy Gradient for Balancing an Inverted Pendulum. **arXiv preprint arXiv:2102.08362**, 2021.

BEARD, Randal W.; MCLAIN, Timothy W. **Small Unmanned Aircraft**: Theory and Practice. 1st. Princeton: Princeton University Press, 2012. P. 143–158.

BI, Yifei; CHEN, Xinyi; XIAO, Caihui. A Deep Reinforcement Learning Approach towards Pendulum Swing-up Problem based on TF-Agents. **arXiv preprint arXiv:2106.09556**, 2021.

BISHOP, Christopher M. **Pattern Recognition and Machine Learning**. 1st. New York: Springer, 2006. P. 256–257.

BONARINI, Andrea; CACCIA, Claudio; LAZARIC, Alessandro; RESTELLI, Marcello. Batch Reinforcement Learning for Controlling a Mobile Wheeled Pendulum Robot. *In*: INTERNATIONAL FEDERATION FOR INFORMATION PROCESSING, 7–10 set. 2008, Milano. ARTIFICIAL Intelligence and Practice II. Boston: Springer, 2008. P. 151–160.

BOUBAKER, Olfa. The inverted Pendulum: A fundamental Benchmark in Control Theory and Robotics. *In*: INTERNATIONAL CONFERENCE ON EDUCATION AND E-LEARNING INNOVATIONS, 1–3 jul. 2012, Sousse, p. 1–6.

BOUBAKER, Olfa. The Inverted Pendulum Benchmark in Nonlinear Control Theory: A Survey. **International Journal of Advanced Robotic Systems**, v. 10, n. 1, 2013.

DURAND, Sylvain; CASTELLANOS, Fermi Guerrero; MARCHAND, Nicolas; SÁNCHEZ, W. Fermín Guerrero. Event-Based Control of the Inverted Pendulum: Swing up and Stabilization. **Journal of Control Engineering and Applied Informatics**, v. 15, n. 3, p. 96–104, 2013.

FERNANDES, Carlos M. C. G.; MARQUES, Pedro M. T.; MARTINS, Ramiro C.; SEABRA, Jorge H. O. Gearbox power loss: Part II: Friction losses in gears. **Tribology International**, v. 88, p. 309–316, 2015.

FUJIMOTO, Scott; HOOF, Herke van; MEGER, David. Addressing Function Approximation Error in Actor-Critic Methods. *In*: 35TH INTERNATIONAL CONFERENCE ON MACHINE LEARNING, 10–15 jul. 2018, Stockholm, p. 1587–1596.

GARCIA, Elena; SANTOS, Pablo Gonzalez de; WIT, Carlos Canudas de. Velocity Dependence in the Cyclic Friction Arising with Gears. **The International Journal of Robotics Research**, v. 21, n. 9, p. 761–771, 2002.

GRASSER, Felix; D'ARRIGO, Aldo; COLOMBI, Silvio; RUFER, Alfred C. JOE: A Mobile, Inverted Pendulum. **IEEE Transactions on Industrial Electronics**, v. 49, n. 1, p. 107–114, 2002.

HASSELT, Hado Van; GUEZ, Arthur; SILVER, David. Deep Reinforcement Learning with Double Q-learning. *In*: THIRTIETH AAAI CONFERENCE ON ARTIFICIAL INTELLIGENCE, Phoenix. PROCEEDINGS of the AAAI Conference on Artificial Intelligence. Palo Alto: AAAI Press, 2016. P. 387–395.

HENDERSON, Peter; ISLAM, Riashat; BACHMAN, Philip; PINEAU, Joelle; PRECUP, Doina; MEGER, David. Deep Reinforcement Learning that Matters. *In*: THIRTY-SECOND AAAI CONFERENCE ON ARTIFICIAL INTELLIGENCE, 2–7 fev. 2018, New Orleans.

HOSOKAWA, Shu; NAKANO, Kazushi. A Reward Allocation Method for Reinforcement Learning in Stabilizing Control of T-inverted Pendulum. *In*: 9TH INTERNATIONAL CONFERENCE ON ELECTRICAL ENGINEERING/ELECTRONICS, COMPUTER, TELECOMMUNICATIONS AND INFORMATION TECHNOLOGY, 16–18 mai. 2012, Phetchaburi, p. 1–4.

HUGHES, Austin; DRURY, Bill. **Electric Motors and Drives**: Fundamentals, Types and Applications. 4th. Waltham: Elsevier, 2013. P. 73–111.

IZZO, Dario; MÄRTENS, Marcus; PAN, Binfeng. A survey on artificial intelligence trends in spacecraft guidance dynamics and control. **Astrodynamics**, v. 3, p. 287–299, 2019.

KARA, Tolgay; EKER, Ilyas. Nonlinear modeling and identification of a DC motor for bidirectional operation with real time experiments. **Energy Conversion and Management**, v. 45, n. 7-8, p. 1087–1106, 2004.

KHALIL, Hassan K. **Nonlinear Systems**. 3rd. Upper Saddle River: Prentice Hall, 1996. P. 1–6.

KIM, Ju-Bong; LIM, Hyun-Kyo; KIM, Chan-Myung; KIM, Min-Suk; HONG, Yong-Geun; HAN, Youn-Hee. Imitation Reinforcement Learning-Based Remote Rotary Inverted Pendulum Control in OpenFlow Network. **IEEE Access**, v. 7, p. 36682–36690, 2019.

KLÖPPELT, Christian; MEYER, Dagmar. Comparison of different Methods for Encoder Speed Signal Filtering exemplified by an Inverted Pendulum. *In*: INTERNATIONAL CONFERENCE ON RESEARCH AND EDUCATION IN MECHATRONICS, 7–8 jun. 2018, Delft, p. 1–6.

KOBER, Jens; BAGNELL, J. Andrew; PETERS, Jan. Reinforcement Learning in Robotics: A Survey. **The International Journal of Robotics Research**, v. 32, n. 11, p. 1238–1274, 2013.

KRISHNAN, Ramu. **Electric Motor Drives**: Modeling, Analysis, and Control. 1st. Upper Saddle River: Prentice Hall, 2001. P. 31–32.

KUMAR, Raj; NIGAM, M. J.; SHARMA, Sudeep; BHAVSAR, Punitkumar. Temporal Difference based Tuning of Fuzzy Logic Controller through Reinforcement Learning to

Control an Inverted Pendulum. **International Journal of Intelligent Systems and Applications**, v. 4, n. 9, p. 15–21, 2012.

LI, Xiaoqian; LIU, Houde; WANG, Xueqian. Solve the inverted pendulum problem base on DQN algorithm. *In*: CHINESE CONTROL AND DECISION CONFERENCE, 3–5 jun. 2019, Nanchang, p. 5115–5120.

LILLICRAP, Timothy P.; HUNT, Jonathan J.; PRITZEL, Alexander; HEESS, Nicolas; EREZ, Tom; TASSA, Yuval; SILVER, David; WIERSTRA, Daan. Continuous Control with Deep Reinforcement Learning. **arXiv preprint arXiv:1509.02971**, 2016.

LIN, Long-Ji. Self-Improving Reactive Agents Based On Reinforcement Learning, Planning and Teaching. **Machine learning**, v. 8, n. 3-4, p. 293–321, 1992.

MAHAJAN, Nayana P.; DESHPANDE, S. B. Study of Nonlinear Behavior of DC Motor Using Modeling and Simulation. **International Journal of Scientific and Research Publications**, v. 3, n. 3, 2013.

MELLATSHAHI, Navid; MOZAFFARI, Saeed; SAIF, Mehrdad; ALIREZAEE, Shahpour. Inverted Pendulum Control with a Robotic Arm using Deep Reinforcement Learning. *In*: INTERNATIONAL SYMPOSIUM ON SIGNALS, CIRCUITS AND SYSTEMS (ISSCS), 15–16 jul. 2021, Iasi, p. 1–6.

MÉNDEZ, Eliana Acurio. Mechanical redesign and control with a PLC of an inverted pendulum. **Congreso de Ciencia y Tecnología ESPE**, v. 10, n. 1, p. 182–187, 2015.

MNIH, Volodymyr *et al.* Human-level control through deep reinforcement learning. **Nature**, v. 518, p. 529–533, 2015.

MORIMOTO, Jun; DOYA, Kenki. Robust Reinforcement Learning. **Neural Computation**, v. 17, n. 2, p. 335–359, 2005.

NAIR, Ashvin; MCGREW, Bob; ANDRYCHOWICZ, Marcin; ZAREMBA, Wojciech; ABBEEL, Pieter. Overcoming Exploration in Reinforcement Learning with Demonstrations. *In*: IEEE INTERNATIONAL CONFERENCE ON ROBOTICS AND AUTOMATION, 21–25 mai. 2018, Brisbane.

OGATA, Katsuhiko. **Modern Control Engineering**. 5th. Upper Saddle River: Prentice Hall, 2010. P. 68–71, 806–807.

PATI, Jyoti Ranjan. **Modeling, Identification and Control of Cart-Pole System**. 2014. Diss. (Mestrado) – National Institute of Technology, Rourkela.

POLYDOROS, Athanasios S.; NALPANTIDIS, Lazaros. Survey of Model-Based Reinforcement Learning: Applications on Robotics. **Journal of Intelligent & Robotic Systems**, v. 86, p. 153–173, 2017.

PONCE, Pedro; MOLINA, Arturo; ALVAREZ, Eugenio. A Review of Intelligent Control Systems Applied to the Inverted-Pendulum Problem. **American Journal of Engineering and Applied Sciences**, v. 7, n. 2, p. 161–207, 2014.

PRIYANKA, K; MARIYAMMAL, A. DC Motor Speed Control Using PWM. **International Journal of Innovative Science and Research Technology**, v. 3, n. 2, 2018.

PURIEL-GIL, Guillermo; YU, Wen; SOSSA, Humberto. Reinforcement Learning Compensation based PD Control for Inverted Pendulum. *In*: 15TH INTERNATIONAL CONFERENCE ON ELECTRICAL ENGINEERING, COMPUTING SCIENCE AND AUTOMATIC CONTROL, 5–7 set. 2018, Mexico City, p. 1–4.

REKDALSBAKKEN, Webjørn. Feedback Control of an Inverted Pendulum with the use of Artificial Intelligence. *In*: INTERNATIONAL CONFERENCE ON COMPUTATIONAL CYBERNETICS, 20 ago. 2006–22 ago. 2005, Talinn, p. 1–6.

SHARMA, Siddharth. Modeling an Inverted Pendulum via Differential Equations and Reinforcement Learning Techniques. **Preprints**, 2020.

SILVER, David; LEVER, Guy; HEESS, Nicolas; DEGRIS, Thomas; WIERSTRA, Daan; RIEDMILLER, Martin. Deterministic Policy Gradient Algorithms. *In*: INTERNATIONAL CONFERENCE ON MACHINE LEARNING, 22–24 jun. 2014, Bejing, p. 387–395.

STEINDÓR, Sæmundsson; HOFMANN, Katja; DEISENROTH, Marc Peter. Meta Reinforcement Learning with Latent Variable Gaussian Processes. **arXiv preprint arXiv:1803.07551**, 2018.

SUTTON, Richard S.; BARTO, Andrew G. **Reinforcement Learning**: An Introduction. 2nd. Cambridge: MIT Press, 2018. P. 48, 73–90, 243–246.

TAKEI, Toshinobu; IMAMURA, Ryoko; YUTA, Shin'ichi. Baggage Transportation and Navigation by a Wheeled Inverted Pendulum Mobile Robot. **IEEE Transactions on Industrial Electronics**, v. 56, n. 10, p. 3985–3994, 2009.

VERMA, Amit; LAMSAL, Kamal; VERMA, Payal. An investigation of skill requirements in artificial intelligence and machine learning job advertisements. **Industry and Higher Education**, v. 36, n. 1, p. 63–73, 2021.

VICHUGOV, V. N.; TSAPKO, G. P.; TSAPKO, S. G. Application of Reinforcement Learning in Control System Development. *In*: THE 9TH RUSSIAN-KOREAN INTERNATIONAL SYMPOSIUM ON SCIENCE AND TECHNOLOGY, 26 jun.–2 jul. 2005, Novosibirsk, p. 732–733.

WATKINS, Christopher John Cornish Hellaby. **Learning from delayed rewards**. 1989. Tese (Doutorado) – King's College.

YANG, Li; SHAMI, Abdallah. On hyperparameter optimization of machine learning algorithms: Theory and practice. **Neurocomputing**, v. 415, p. 295–316, 2020.

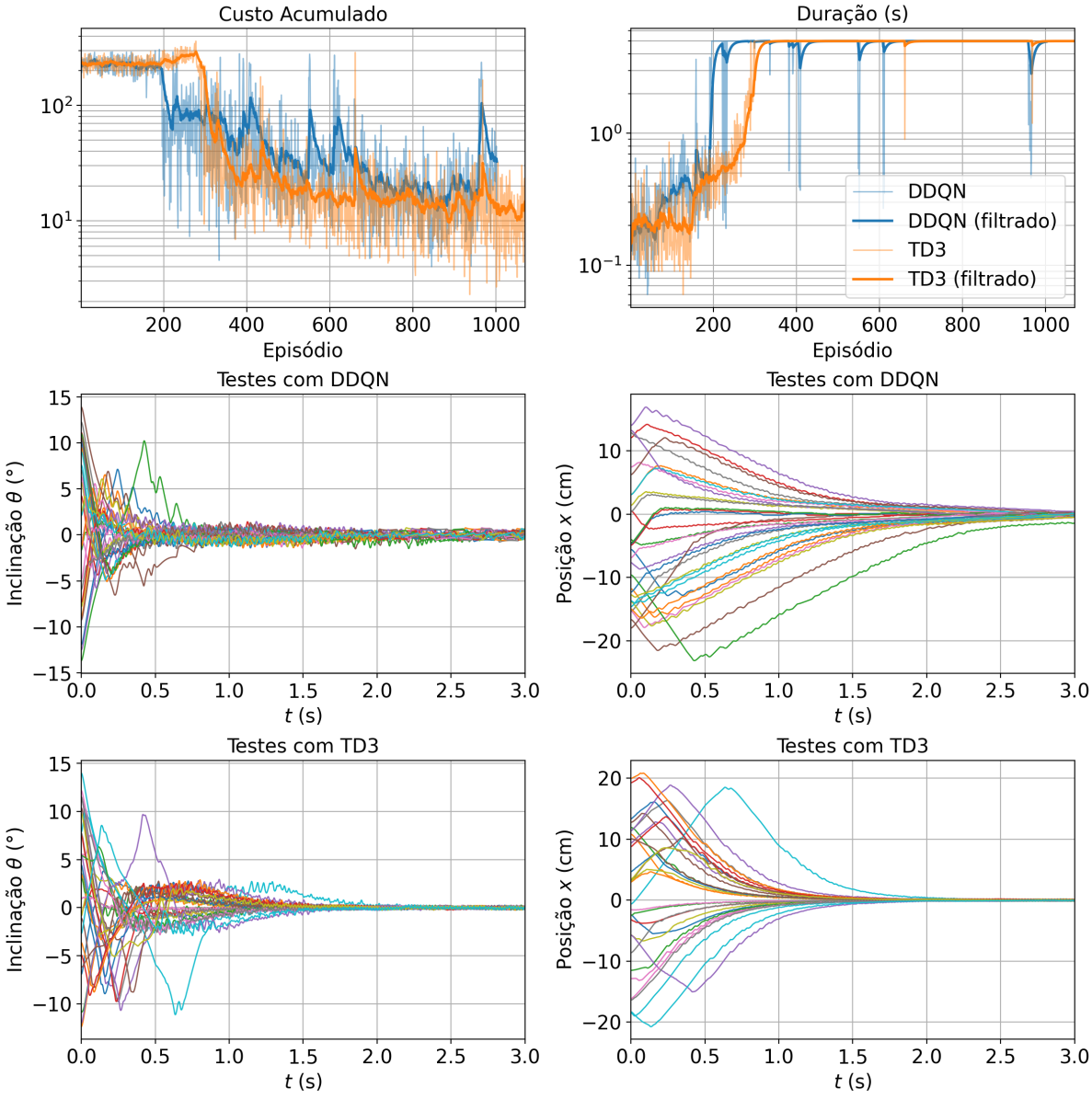
YOUNIS, Wael; ABDELATI, Mohammed. Design and Implementation of an Experimental Segway Model. **AIP Conference Proceedings**, v. 1107, n. 1, 2009.

ZHU, Z. Q. A Simple Method for Measuring Cogging Torque in Permanent Magnet Machines. *In*: 2009 IEEE POWER & ENERGY SOCIETY GENERAL MEETING, 26–30 jul. 2009, Calgary, p. 1–4.

APÊNDICE A – CÓDIGO IMPLEMENTADO

O código disponibilizado na página seguinte foi testado no ambiente *Kaggle*, com tempo de execução de aproximadamente oito horas. Este contém a implementação, de acordo com este trabalho, do modelo de pêndulo invertido e treinamento dos métodos *Double Deep Q-Network* e *Twin Delayed Deep Deterministic Policy Gradient*. No final de sua execução, este código exibe gráficos referentes aos treinamentos e testes dos controladores resultantes. Estas ilustrações foram elaboradas de modo similar aos gráficos exibidos no capítulo 4, conforme exemplo da Figura 38.

Figura 38 – Exemplo de resultado do código disponibilizado.



Fonte: Elaboração própria.

```
import numpy as np
import matplotlib.pyplot as plt
import math
from scipy.integrate import odeint
import ipywidgets
import random
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torch as T
import copy

if T.cuda.is_available():
    device = T.device('cuda:0')
else:
    device = T.device('cpu')
    print('Using CPU')
class Model():
    def __init__(self, rtol=1e-12, atol=1e-12, n=100, x=0.0, dx=0.0, p=0.0, dp=0.0, p_lim=20.0, x_lim
        =0.5, Ts=0.01, lambda_x=1.0, lambda_p=1.0, l=1.67e-3, l=0.1, r=0.02, m=0.5, J=0.02e-3, b
        =272e-6, Ke=0.686, Kt=0.082, R=7.4, L=7.45e-3):
        self.l = l # Body moment of inertia (kgm2)
        self.l = l # Pole lenght (m)
        self.r = r # Whell radius (m)
        self.m = m # Pole mass (kg)
        self.J = J # Wheel moment of inertia (kgm2)
        self.b = b # Motor friction coefficient (Nms)
        self.Kt = Kt # Motor torque constant (Nm/A)
        self.Ke = Ke # Motor speed constant (Vs)
        self.R = R # Motor resistance (R)
        self.L = L # Motor inductance (H)
        self.g = 9.80665 # Gravity (m/s2)
        self.Ax = ((self.r*self.m*self.l)**2)*self.g
        self.Bx = (self.l + self.m*(self.l**2))*self.r*self.Kt
        self.Cx = (self.l + self.m*(self.l**2))*self.b
        self.Dx = (self.l + self.m*(self.l**2))*(self.r**2)*self.m*self.l
        self.Ap = (self.J + (self.r**2)*self.m)*self.m*self.g*self.l
        self.Bp = self.m*self.l*self.r*self.Kt
        self.Cp = self.m*self.l*self.b
        self.Dp = (self.m*self.l*self.r)**2
        self.E = (self.J + (self.r**2)*self.m)*(self.m*(self.l**2) + self.l)
        self.F = (self.r*self.m*self.l)**2
        self.Ts = Ts # Iteration Period (s)
        self.n = n # Steps per Period
        self.T = self.Ts / self.n # Simulation Period (s)
        self.rtol = rtol
        self.atol = atol
```

```
        self.p_lim = p_lim*math.pi/180.0
        self.x_lim = x_lim
        self.lambda_x = lambda_x
        self.lambda_p = lambda_p
        self.v = 0.0 # (V)
        self.x = x # (m)
        self.dx = dx # (m/s)
        self.p = p # (rad)
        self.dp = dp # (rad/s)
        self.i = 0.0 # (A)
        self.t = 0.0 # (s)
        self.v_list = [self.v]
        self.x_list = [self.x]
        self.dx_list = [self.dx]
        self.p_list = [self.p]
        self.dp_list = [self.dp]
        self.i_list = [self.i]
        self.t_list = [self.t]
    def reset_var(self, x=0.0, dx=0.0, p=0.0, dp=0.0):
        self.v = 0.0
        self.x = x
        self.dx = dx
        self.p = p
        self.dp = dp
        self.i = 0.0
        self.t = 0.0
        self.x_list = [self.x]
        self.dx_list = [self.dx]
        self.p_list = [self.p]
        self.dp_list = [self.dp]
        self.i_list = [self.i]
        self.t_list = [self.t]
        self.v_list = [self.v]
    def derivative_L0(self, t, y):
        p, dp, x, dx = y
        i = (self.v – self.Ke*dx/self.r) / self.R
        co = math.cos(p)
        si = math.sin(p)
        D = self.E – self.F*(co**2)
        return [dp, (self.Ap*si – co*(self.Bp*i – self.Cp*dx + self.Dp*si*(dp**2))) / D, dx, (–self.Ax*si*co
            + self.Bx*i – self.Cx*dx + self.Dx*si*(dp**2)) / D]
    def iteration_odeint(self):
        t = np.linspace(self.t, self.t + self.Ts, 2)
        sol = odeint(self.derivative_L0, [self.p, self.dp, self.x, self.dx], t, tfirst=True, rtol=self.rtol, atol=
            self.atol)
        self.t = t[–1]
        self.p = sol[–1, 0]
```

```
            self.dp = sol[-1, 1]
            self.x = sol[-1, 2]
            self.dx = sol[-1, 3]
        def iteration_odeint_n(self):
            t = np.linspace(self.t, self.t + self.Ts, self.n + 1)
            sol = odeint(self.derivative_L0, [self.p, self.dp, self.x, self.dx], t, tfirst=True, rtol=self.rtol, atol=
                self.atol)
            self.t = t[-1]
            self.p = sol[-1, 0]
            self.dp = sol[-1, 1]
            self.x = sol[-1, 2]
            self.dx = sol[-1, 3]
            self.i = (self.v - self.Ke*self.dx/self.r) / self.R
            self.t_list += list(t[1:])
            self.p_list += list(sol[1:, 0])
            self.dp_list += list(sol[1:, 1])
            self.x_list += list(sol[1:, 2])
            self.dx_list += list(sol[1:, 3])
            self.i_list += list((self.v - self.Ke*sol[1:, 3]/self.r) / self.R)
            self.v_list += list(np.ones(self.n)*self.v)
        def get_x(self):
            return np.array(self.x_list)
        def get_dx(self):
            return np.array(self.dx_list)
        def get_p_180(self):
            return np.array(self.p_list)*180.0/math.pi
        def get_dp(self):
            return np.array(self.dp_list)*180.0/math.pi
        def get_i(self):
            return np.array(self.i_list)
        def get_t(self):
            return np.array(self.t_list)
        def get_v(self):
            return np.array(self.v_list)
        def isDone(self):
            if self.p > self.p_lim or self.p < -self.p_lim or self.x > self.x_lim or self.x < -self.x_lim:
                return True
            else:
                return False
        def getState(self):
            return np.array([self.x, self.dx, self.p, self.dp])
        def getReward(self, x, p):
            return -self.lambda_p*np.abs(p) - self.lambda_x*np.abs(x)
class DDQN_NN5(nn.Module):
    def __init__(self, device, lr, input_size, output_size, l1=50, l2=50, l3=50, l4=50, l5=50):
        super(DDQN_NN5, self).__init__()
        self.fc1 = nn.Linear(input_size, l1)
```

```
        self.fc2 = nn.Linear(l1, l2)
        self.fc3 = nn.Linear(l2, l3)
        self.fc4 = nn.Linear(l3, l4)
        self.fc5 = nn.Linear(l4, l5)
        self.fc6 = nn.Linear(l5, output_size)
        self.optimizer = optim.Adam(self.parameters(), lr=lr)
        self.loss = nn.MSELoss()
        self.to(device)
    def forward(self, input_data):
        layer1 = F.relu(self.fc1(input_data))
        layer2 = F.relu(self.fc2(layer1))
        layer3 = F.relu(self.fc3(layer2))
        layer4 = F.relu(self.fc4(layer3))
        layer5 = F.relu(self.fc5(layer4))
        return self.fc6(layer5)
class Agent_DDQN(object):
    def __init__(self, device, gamma, lr, max_steps, max_v, n_actions, model, final_reward, n_states
        =4, eps_start=1.0, eps_stop=0.1, eps_dec=1e−3, buffer_size=1000, batch_size=32,
        att_target_period=1000, l1=50, l2=0, l3=0, l4=0, l5=0):
        self.device = device
        self.gamma = gamma
        self.lr = lr
        self.actions_v = np.linspace(−max_v, max_v, n_actions)
        self.actions = np.arange(n_actions)
        self.model = model
        self.eps = eps_start
        self.eps_stop = eps_stop
        self.eps_dec = eps_dec
        self.step = 0
        self.att_target_period = att_target_period
        self.loss_lp = 0.0
        self.loss_f = 0.99
        self.final_reward_gamma = final_reward / (1 − gamma)
        self.buffer_size = np.int64(buffer_size)
        self.buffer_counter = 0
        self.batch_size = batch_size
        self.buffer_state = np.zeros((self.buffer_size, n_states), dtype=np.float32)
        self.buffer_state_ = np.zeros((self.buffer_size, n_states), dtype=np.float32)
        self.buffer_action = np.zeros(self.buffer_size, dtype=np.int32)
        self.buffer_reward = np.zeros(self.buffer_size, dtype=np.float32)
        self.buffer_done = np.zeros(self.buffer_size, dtype=bool)
        self.step_hist, self.eps_hist, self.score_hist, self.t_max_hist, self.loss_hist = [], [], [], [], []
        self.run_hist, self.t_hist, self.x_hist, self.dx_hist, self.p_hist, self.dp_hist, self.v_hist = [], [], [], [],
            [], [], []
        self.q_eval = DDQN_NN5(device=device, lr=lr, input_size=n_states, output_size=n_actions, l1
            =l1, l2=l2, l3=l3, l4=l4, l5=l5)
```

```
        self.q_targ = DDQN_NN5(device=device, lr=lr, input_size=n_states, output_size=n_actions, l1
            =l1, l2=l2, l3=l3, l4=l4, l5=l5)
    def best_action(self, state):
        state = T.tensor(state, dtype=T.float32).to(self.device)
        action = self.q_eval.forward(state)
        return T.argmax(action).item()
    def best_action_v(self, state):
        return self.actions_v[self.best_action(state)]
    def get_action(self, state):
        if np.random.random() > self.eps:
            return self.best_action(state)
        else:
            return np.random.choice(self.actions)
    def get_action_v(self, state):
        return self.actions_v[self.get_action(state)]
    def buffer_save(self, state, action, reward, state_, done):
        index = self.buffer_counter % self.buffer_size
        self.buffer_state[index] = state
        self.buffer_state_[index] = state_
        self.buffer_action[index] = np.where(self.actions_v == action)[0]
        self.buffer_reward[index] = reward
        self.buffer_done[index] = done
        self.buffer_counter += 1
    def buffer_load(self):
        max_index = min(self.buffer_counter, self.buffer_size)
        batch = np.random.choice(max_index, self.batch_size, replace=False)
        states = T.tensor(self.buffer_state[batch]).to(self.device)
        states_ = T.tensor(self.buffer_state_[batch]).to(self.device)
        actions = T.tensor(self.buffer_action[batch], dtype=T.long).to(self.device)
        rewards = T.tensor(self.buffer_reward[batch]).to(self.device)
        dones = T.tensor(self.buffer_done[batch]).to(self.device)
        return states, states_, actions, rewards, dones
    def save_step(self, score):
        self.step_hist.append(self.step)
        self.eps_hist.append(self.eps)
        self.score_hist.append(score)
        self.t_max_hist.append(self.model.t)
        self.loss_hist.append(self.loss_lp)
    def learn(self):
        if self.buffer_counter < self.batch_size:
            return
        self.step += 1
        if self.step % self.att_target_period == 0:
            self.q_targ.load_state_dict(self.q_eval.state_dict())
        self.q_eval.optimizer.zero_grad()
        states, states_, actions, rewards, dones = self.buffer_load()
        batch_indexes = np.arange(self.batch_size)
```

```
            q_eval = self.q_eval.forward(states)[batch_indexes, actions]
            q_eval_ = self.q_eval.forward(states_)
            actions_ = T.argmax(q_eval_, dim=1)
            q_targ = self.q_targ.forward(states_)
            q_targ = q_targ[batch_indexes, actions_]
            q_targ[dones] = self.final_reward_gamma
            q_eval_update = rewards + self.gamma*q_targ
            loss = self.q_eval.loss(q_eval_update, q_eval).to(self.device)
            loss.backward()
            self.q_eval.optimizer.step()
            self.loss_lp = self.loss_f*self.loss_lp + (1-self.loss_f)*loss.item()
            self.eps = self.eps - self.eps_dec if self.eps > self.eps_stop else self.eps_stop
def train_DDQN(device, max_steps=1000, sim_time=5.0, max_v=12.0, n_actions=13, model=Model(),
        x0=0.2, dx0=0.0, p0=14.0, dp0=0.0, agent=None, gamma=0.99, lr=1e-4, eps_start=1.0,
        eps_stop=0.01, eps_dec=1e-3, buffer_size=2000, batch_size=128, att_target_period=100, l1=50,
        l2=0, l3=0, l4=0, l5=0, score_limit=25, eval_interval=50, eval_runs=10):
    final_reward = model.getReward(model.x_lim, model.p_lim)
    if agent == None:
        agent = Agent_DDQN(device=device, gamma=gamma, lr=lr, max_steps=max_steps, max_v=
            max_v, n_actions=n_actions, model=model, final_reward=final_reward, eps_start=
            eps_start, eps_stop=eps_stop, eps_dec=eps_dec, buffer_size=buffer_size, batch_size=
            batch_size, att_target_period=att_target_period, l1=l1, l2=l2, l3=l3, l4=l4, l5=l5)
    it = int(sim_time/model.Ts)
    lbl = ipywidgets.Label(value='')
    display(lbl)
    flt = 0.9
    t_flt = 0.0
    score_flt = final_reward * it
    run = 0
    print('Training DDQN')
    agent_copy = copy.deepcopy(agent)
    score_eval = -score_limit
    while agent.step < max_steps:
        run += 1
        x = random.uniform(-x0, x0)
        dx = random.uniform(-dx0, dx0)
        p = random.uniform(-p0, p0)*math.pi/180.0
        dp = random.uniform(-dp0, dp0)*math.pi/180.0
        agent.model.reset_var(x=x, dx=dx, p=p, dp=dp)
        score = 0.0
        for i in range(it):
            state = agent.model.getState()
            if state[0] < 0:
                state = -state
                sgn = -1
            else:
                sgn = 1
```

```
        action = agent.get_action_v(state)
        agent.model.v = sgn*action
        agent.model.iteration_odeint()
        state_ = agent.model.getState()
        if state_[0] < 0:
            state_ = -state_
        done = agent.model.isDone()
        reward = agent.model.getReward(agent.model.x, agent.model.p)
        agent.buffer_save(state, action, reward, state_, done)
        agent.learn()
        if done:
            score += reward * (it - i)
            break
        else:
            score += reward
    agent.save_step(score)
    t_flt = t_flt*flt + agent.model.t*(1-flt)
    score_flt = score_flt*flt + score*(1-flt)
    lbl.value = "{:.1%}, step: {:,.0f}/{:,.0f}, run: {:,}, t: {:.2f}, score: {:.2f}".format(agent.step/
        max_steps, agent.step, max_steps, run, t_flt, score_flt)
    if (run % eval_interval == 0) and (score_flt > (-score_limit)):
        scores = []
        for j in range(eval_runs):
            x = random.uniform(-x0, x0)
            dx = random.uniform(-dx0, dx0)
            p = random.uniform(-p0, p0)*math.pi/180.0
            dp = random.uniform(-dp0, dp0)*math.pi/180.0
            agent.model.reset_var(x=x, dx=dx, p=p, dp=dp)
            score = 0.0
            for i in range(it):
                state = agent.model.getState()
                if state[0] < 0:
                    state = -state
                    sgn = -1
                else:
                    sgn = 1
                action = agent.best_action_v(state)
                agent.model.v = sgn*action
                agent.model.iteration_odeint()
                done = agent.model.isDone()
                reward = agent.model.getReward(agent.model.x, agent.model.p)
                if done:
                    score += reward * (it - i)
                    break
                else:
                    score += reward
            scores.append(score)
```

```
            if np.mean(scores) > score_eval:
                score_eval = np.mean(scores)
                agent_copy = copy.deepcopy(agent)
    print('Best␣DDQN␣Score:', score_eval)
    return agent_copy, agent
class TD3_NN4(nn.Module):
    def __init__(self, device, lr, input_size, output_size, l1=50, l2=50, l3=50, l4=50):
        super(TD3_NN4, self).__init__()
        self.fc1 = nn.Linear(input_size, l1)
        self.fc2 = nn.Linear(l1, l2)
        self.fc3 = nn.Linear(l2, l3)
        self.fc4 = nn.Linear(l3, l4)
        self.fc5 = nn.Linear(l4, output_size)
        self.optimizer = optim.Adam(self.parameters(), lr=lr)
        self.loss = nn.MSELoss()
        self.to(device)
    def forward(self, input_data):
        layer1 = F.relu(self.fc1(input_data))
        layer2 = F.relu(self.fc2(layer1))
        layer3 = F.relu(self.fc3(layer2))
        layer4 = F.relu(self.fc4(layer3))
        return self.fc5(layer4)
class Agent_TD3(object):
    def __init__(self, device, gamma, lr_actor, lr_crit, model, final_reward, n_states=4, n_actions=1,
        max_action=12, action_norm=0.02, buffer_size=1000, batch_size=32, tau=0.005, noise_std_a
        =0.01, noise_std_a_=0.01, noise_clamp_factor=5.0, update_actor_interval=2, random_steps
        =1000, l1=50, l2=50, l3=0, l4=0):
        self.device = device
        self.gamma = gamma
        self.lr_actor = lr_actor
        self.lr_crit = lr_crit
        self.model = model
        self.step = 0
        self.noise_std_a = noise_std_a
        self.noise_std_a_ = noise_std_a_
        self.noise_clamp_factor = noise_clamp_factor
        self.random_steps = random_steps
        self.max_action = max_action
        self.update_actor_iter = update_actor_interval
        self.loss_actor_lp = 0.0
        self.loss_crit1_lp = 0.0
        self.loss_crit2_lp = 0.0
        self.loss_f = 0.99
        self.final_reward_gamma = final_reward / (1 − gamma)
        self.action_norm = action_norm
        self.buffer_size = np.int64(buffer_size)
        self.buffer_counter = 0
```

```
        self.batch_size = batch_size
        self.buffer_state = np.zeros((self.buffer_size, n_states), dtype=np.float32)
        self.buffer_state_ = np.zeros((self.buffer_size, n_states), dtype=np.float32)
        self.buffer_action = np.zeros(self.buffer_size, dtype=np.float32)
        self.buffer_reward = np.zeros(self.buffer_size, dtype=np.float32)
        self.buffer_done = np.zeros(self.buffer_size, dtype=bool)
        self.step_hist, self.score_hist, self.t_max_hist, self.loss_hist, self.loss_crit1_hist, self.
            loss_crit2_hist = [], [], [], [], [], []
        self.run_hist, self.t_hist, self.x_hist, self.dx_hist, self.p_hist, self.dp_hist, self.v_hist = [], [], [], [],
            [], [], []
        self.actor_eval = TD3_NN4(device=device, lr=lr_actor, input_size=n_states, output_size=
            n_actions, l1=l1, l2=l2, l3=l3, l4=l4)
        self.actor_targ = TD3_NN4(device=device, lr=lr_actor, input_size=n_states, output_size=
            n_actions, l1=l1, l2=l2, l3=l3, l4=l4)
        self.crit1_eval = TD3_NN4(device=device, lr=lr_crit, input_size=n_states+n_actions,
            output_size=1, l1=l1, l2=l2, l3=l3, l4=l4)
        self.crit1_targ = TD3_NN4(device=device, lr=lr_crit, input_size=n_states+n_actions,
            output_size=1, l1=l1, l2=l2, l3=l3, l4=l4)
        self.crit2_eval = TD3_NN4(device=device, lr=lr_crit, input_size=n_states+n_actions,
            output_size=1, l1=l1, l2=l2, l3=l3, l4=l4)
        self.crit2_targ = TD3_NN4(device=device, lr=lr_crit, input_size=n_states+n_actions,
            output_size=1, l1=l1, l2=l2, l3=l3, l4=l4)
        self.tau = 1
        self.update_NN()
        self.tau = tau
    def best_action_v(self, state):
        state = T.tensor(state, dtype=T.float32).to(self.device)
        action = self.actor_eval.forward(state).to(self.device)
        action = T.tanh(action)*self.max_action
        return action.item()
    def get_action_v(self, state):
        if self.step < self.random_steps:
            action = T.tensor(np.random.uniform(-self.max_action, self.max_action)).to(self.device)
        else:
            state = T.tensor(state, dtype=T.float32).to(self.device)
            action = self.actor_eval.forward(state).to(self.device)
            action = T.tanh(action)*self.max_action
        action_noise = action + T.tensor(np.random.normal(scale=self.noise_std_a), dtype=T.float32).
            to(self.device)*self.max_action
        action_noise = T.clamp(action_noise, -self.max_action, self.max_action)
        return action_noise.item()
    def buffer_save(self, state, action, reward, state_, done):
        index = self.buffer_counter % self.buffer_size
        self.buffer_state[index] = state
        self.buffer_state_[index] = state_
        self.buffer_action[index] = action
        self.buffer_reward[index] = reward
```

```
        self.buffer_done[index] = done
        self.buffer_counter += 1
    def buffer_load(self):
        max_index = min(self.buffer_counter, self.buffer_size)
        batch = np.random.choice(max_index, self.batch_size, replace=False)
        states = T.tensor(self.buffer_state[batch]).to(self.device)
        states_ = T.tensor(self.buffer_state_[batch]).to(self.device)
        actions = T.tensor(self.buffer_action[batch], dtype=T.long).to(self.device)
        rewards = T.tensor(self.buffer_reward[batch]).to(self.device)
        dones = T.tensor(self.buffer_done[batch]).to(self.device)
        return states, states_, actions, rewards, dones
    def save_step(self, score):
        self.step_hist.append(self.step)
        self.score_hist.append(score)
        self.t_max_hist.append(self.model.t)
        self.loss_hist.append(self.loss_actor_lp)
        self.loss_crit1_hist.append(self.loss_crit1_lp)
        self.loss_crit2_hist.append(self.loss_crit2_lp)
    def update_NN(self):
        actor_eval_param = dict(self.actor_eval.named_parameters())
        actor_targ_param = dict(self.actor_targ.named_parameters())
        crit1_eval_param = dict(self.crit1_eval.named_parameters())
        crit1_targ_param = dict(self.crit1_targ.named_parameters())
        crit2_eval_param = dict(self.crit2_eval.named_parameters())
        crit2_targ_param = dict(self.crit2_targ.named_parameters())
        for i in actor_targ_param:
            actor_targ_param[i] = self.tau*actor_eval_param[i].clone() + (1-self.tau)*actor_targ_param
                [i].clone()
        for i in crit1_targ_param:
            crit1_targ_param[i] = self.tau*crit1_eval_param[i].clone() + (1-self.tau)*crit1_targ_param[i].
                clone()
        for i in crit2_targ_param:
            crit2_targ_param[i] = self.tau*crit2_eval_param[i].clone() + (1-self.tau)*crit2_targ_param[i].
                clone()
        self.actor_targ.load_state_dict(actor_targ_param)
        self.crit1_targ.load_state_dict(crit1_targ_param)
        self.crit2_targ.load_state_dict(crit2_targ_param)
    def learn(self):
        if self.buffer_counter < self.batch_size:
            return
        self.step += 1
        states, states_, actions, rewards, dones = self.buffer_load()
        actions_ = T.tanh(self.actor_targ.forward(states_))*self.max_action
        noise_ = T.tensor(np.random.normal(scale=self.noise_std_a_, size=self.batch_size), dtype=T.
            float32).to(self.device)
        noise_ = T.clamp(noise_, -self.noise_clamp_factor*self.noise_std_a_, self.noise_clamp_factor
            *self.noise_std_a_).unsqueeze(1)
```

```
            actions_noise_ = actions_ + noise_*self.max_action
            actions_noise_ = T.clamp(actions_noise_, -self.max_action, self.max_action)

            actions_noise_states_ = T.cat([actions_noise_*self.action_norm, states_], dim=1)
            actions_states = T.cat([actions.unsqueeze(1)*self.action_norm, states], dim=1)
            q1_ = self.crit1_targ.forward(actions_noise_states_)
            q2_ = self.crit2_targ.forward(actions_noise_states_)
            q1 = self.crit1_eval.forward(actions_states)
            q2 = self.crit2_eval.forward(actions_states)

            rewards = rewards.unsqueeze(1)
            q1_[dones] = self.final_reward_gamma
            q2_[dones] = self.final_reward_gamma
            q_target = rewards + self.gamma*T.min(q1_, q2_)

            self.crit1_eval.optimizer.zero_grad()
            self.crit2_eval.optimizer.zero_grad()
            q1_loss = F.mse_loss(q_target, q1)
            q2_loss = F.mse_loss(q_target, q2)
            q_loss = q1_loss + q2_loss
            q_loss.backward()
            self.crit1_eval.optimizer.step()
            self.crit2_eval.optimizer.step()

            if self.step % self.update_actor_iter == 0:
                self.actor_eval.optimizer.zero_grad()
                actor_actions = T.tanh(self.actor_eval.forward(states))*self.max_action
                actor_actions_states = T.cat([actor_actions*self.action_norm, states], dim=1)
                actor_loss = self.crit1_eval.forward(actor_actions_states)
                actor_loss = -T.mean(actor_loss)
                actor_loss.backward()
                self.actor_eval.optimizer.step()
                self.update_NN()
                self.loss_actor_lp = self.loss_f*self.loss_actor_lp + (1-self.loss_f)*actor_loss.item()
                self.loss_crit1_lp = self.loss_f*self.loss_crit1_lp + (1-self.loss_f)*q1_loss.item()
                self.loss_crit2_lp = self.loss_f*self.loss_crit2_lp + (1-self.loss_f)*q2_loss.item()
def train_TD3(device, max_steps=1000, sim_time=5.0, max_v=12.0, model=Model(), x0=0.2, dx0=0.0,
        p0=14.0, dp0=0.0, agent=None, gamma=0.99, lr=1e-4, buffer_size=2000, batch_size=32, tau
        =0.005, noise_std_a=0.01, noise_std_a_=0.01, noise_clamp_factor=5.0, update_actor_interval=2,
        random_steps=1e3, l1=50, l2=50, l3=0, l4=0, score_limit=25, eval_interval=50, eval_runs=10):
    final_reward = model.getReward(model.x_lim, model.p_lim)
    if agent == None:
        agent = Agent_TD3(device=device, gamma=gamma, lr_actor=lr, lr_crit=lr, model=model,
            final_reward=final_reward, max_action=max_v, buffer_size=buffer_size, batch_size=
            batch_size, tau=tau, noise_std_a=noise_std_a, noise_std_a_=noise_std_a_,
            noise_clamp_factor=noise_clamp_factor, update_actor_interval=update_actor_interval,
            random_steps=random_steps, l1=l1, l2=l2, l3=l3, l4=l4)
```

```
it = int(sim_time/model.Ts)
lbl = ipywidgets.Label(value='')
display(lbl)
flt = 0.9
t_flt = 0.0
score_flt = final_reward * it
run = 0
print('Training␣TD3')
agent_copy = copy.deepcopy(agent)
score_eval = -score_limit
while agent.step < max_steps:
    run += 1
    x = random.uniform(-x0, x0)
    dx = random.uniform(-dx0, dx0)
    p = random.uniform(-p0, p0)*math.pi/180.0
    dp = random.uniform(-dp0, dp0)*math.pi/180.0
    agent.model.reset_var(x=x, dx=dx, p=p, dp=dp)
    score = 0.0
    for i in range(it):
        state = agent.model.getState()
        if state[0] < 0:
            state = -state
            sgn = -1
        else:
            sgn = 1
        action = agent.get_action_v(state)
        agent.model.v = sgn*action
        agent.model.iteration_odeint()
        state_ = agent.model.getState()
        if state_[0] < 0:
            state_ = -state_
        done = agent.model.isDone()
        reward = agent.model.getReward(agent.model.x, agent.model.p)
        agent.buffer_save(state, action, reward, state_, done)
        agent.learn()
        if done:
            score += reward * (it - i)
            break
        else:
            score += reward
    agent.save_step(score)
    t_flt = t_flt*flt + agent.model.t*(1-flt)
    score_flt = score_flt*flt + score*(1-flt)
    lbl.value = "{:.1%},␣step:␣{:,.0f}/{:,.0f},␣run:␣{:,},␣t:␣{:.2f},␣score:␣{:.2f}".format(agent.step/
        max_steps, agent.step, max_steps, run, t_flt, score_flt)
    if (run % eval_interval == 0) and (score_flt > (-score_limit)):
        scores = []
```

```
            for j in range(eval_runs):
                x = random.uniform(−x0, x0)
                dx = random.uniform(−dx0, dx0)
                p = random.uniform(−p0, p0)∗math.pi/180.0
                dp = random.uniform(−dp0, dp0)∗math.pi/180.0
                agent.model.reset_var(x=x, dx=dx, p=p, dp=dp)
                score = 0.0
                for i in range(it):
                    state = agent.model.getState()
                    if state[0] < 0:
                        state = −state
                        sgn = −1
                    else:
                        sgn = 1
                    action = agent.best_action_v(state)
                    agent.model.v = sgn∗action
                    agent.model.iteration_odeint()
                    done = agent.model.isDone()
                    reward = agent.model.getReward(agent.model.x, agent.model.p)
                    if done:
                        score += reward ∗ (it − i)
                        break
                    else:
                        score += reward
                scores.append(score)
            if np.mean(scores) > score_eval:
                score_eval = np.mean(scores)
                agent_copy = copy.deepcopy(agent)
    print('Best␣TD3␣Score:', score_eval)
    return agent_copy, agent
def test_agent(agt, size=1, sim_time=2.0, x0=0.2, dx0=0.0, p0=14.0, dp0=0.0):
    it = int(sim_time/agt.model.Ts)
    models = []
    for s in range(size):
        x = random.uniform(−x0, x0)
        dx = random.uniform(−dx0, dx0)
        p = random.uniform(−p0, p0)∗math.pi/180.0
        dp = random.uniform(−dp0, dp0)∗math.pi/180.0
        modelo = Model(x=x, dx=dx, p=p, dp=dp)
        models.append(modelo)
        for j in range(it):
            state = modelo.getState()
            if state[0] < 0:
                state = −state
                sgn = −1
            else:
                sgn = 1
```

```
            action = agt.best_action_v(state)
            modelo.v = sgn*action
            modelo.iteration_odeint_n()
            if modelo.isDone():
                break
    return models
def filter(x, a=0.9):
    y_list = []
    y = x[0]
    for i in range(len(x)):
        y = y*a + x[i]*(1-a)
        y_list.append(y)
    return y_list
agt_best_DDQN, agt_hist_DDQN = train_DDQN(device=device, max_steps=4e5, lr=3e-6, l1=3000, l2
    =2000, l3=1000, l4=500, l5=250, n_actions=9, buffer_size=19e3, batch_size=128,
    att_target_period=400, score_limit=25, eval_interval=5, eval_runs=200)
agt_best_TD3, agt_hist_TD3 = train_TD3(device=device, max_steps=4e5, buffer_size=4e5, lr=8e-5, l1
    =2000, l2=1000, l3=500, l4=250, noise_std_a=0.01, noise_std_a_=0.12, random_steps=3e3,
    batch_size=128, score_limit=25, eval_interval=5, eval_runs=200)
models_DDQN = test_agent(agt=agt_best_DDQN, size=30, sim_time=3.0)
models_TD3 = test_agent(agt=agt_best_TD3, size=30, sim_time=3.0)
agts = [agt_hist_DDQN, agt_hist_TD3]
a = 0.9
transparency=0.5
fig, ((ax1, ax2), (ax3, ax4), (ax5, ax6)) = plt.subplots(3, 2)
fig.set_size_inches(12, 12)
fig.set_dpi(150)
font_size = 15
linewidth = 1
linewidth_filtered = 2
colors = ['C0', 'C1']
method = ['DDQN', 'TD3']
max_ep = 0
for i in range(len(agts)):
    if len(agts[i].score_hist) > max_ep:
        max_ep = len(agts[i].score_hist)
for i in range(len(agts)):
    x = np.arange(1, len(agts[i].score_hist) + 1)
    ax1.plot(x, -np.array(agts[i].score_hist), linewidth=linewidth, alpha=transparency, color=colors[i])
    ax1.plot(x, -np.array(filter(x=agts[i].score_hist, a=a)), linewidth=linewidth_filtered, color=colors[i])
    ax1.set_yscale('log')
    ax1.set_title('Custo␣Acumulado', fontsize=font_size);
    ax1.set_xlabel('Episódio', fontsize=font_size);
    ax1.set_xlim([1, max_ep])
    ax1.tick_params(axis='x', labelsize=font_size)
    ax1.tick_params(axis='y', labelsize=font_size)
    ax1.grid(True, which="both")
```

```
    ax2.plot(x, agts[i].t_max_hist, linewidth=linewidth, alpha=transparency, color=colors[i], label=
        method[i])
    ax2.plot(x, filter(x=agts[i].t_max_hist, a=a), linewidth=linewidth_filtered, color=colors[i], label=
        method[i]+' (filtrado)')
    ax2.set_yscale('log')
    ax2.set_title('Duração (s)', fontsize=font_size);
    ax2.set_xlabel('Episódio', fontsize=font_size);
    ax2.set_xlim([1, max_ep])
    ax2.tick_params(axis='x', labelsize=font_size)
    ax2.tick_params(axis='y', labelsize=font_size)
    ax2.grid(True, which="both")
    ax2.legend(loc='lower right', fontsize=font_size)
max_t_DDQN = 0.0
max_t_TD3 = 0.0
for i in range(len(models_DDQN)):
    if max(models_DDQN[i].get_t()) > max_t_DDQN:
        max_t_DDQN = max(models_DDQN[i].get_t())
    ax3.plot(models_DDQN[i].get_t(), models_DDQN[i].get_p_180(), linewidth=linewidth)
    ax3.set_title('Testes com DDQN', fontsize=font_size);
    ax3.set_ylabel(r'Inclinação $\theta$ (°)', fontsize=font_size);
    ax3.set_xlabel(r'$t$ (s)', fontsize=font_size);
    ax3.set_xlim([0, max_t_DDQN])
    ax3.tick_params(axis='x', labelsize=font_size)
    ax3.tick_params(axis='y', labelsize=font_size)
    ax3.grid(True, which='both')

    ax4.plot(models_DDQN[i].get_t(), models_DDQN[i].get_x()*100, linewidth=linewidth)
    ax4.set_title('Testes com DDQN', fontsize=font_size);
    ax4.set_ylabel(r'Posição $x$ (cm)', fontsize=font_size);
    ax4.set_xlabel(r'$t$ (s)', fontsize=font_size);
    ax4.set_xlim([0, max_t_DDQN])
    ax4.tick_params(axis='x', labelsize=font_size)
    ax4.tick_params(axis='y', labelsize=font_size)
    ax4.grid(True, which='both')

    if max(models_TD3[i].get_t()) > max_t_TD3:
        max_t_TD3 = max(models_TD3[i].get_t())
    ax5.plot(models_TD3[i].get_t(), models_TD3[i].get_p_180(), linewidth=linewidth)
    ax5.set_title('Testes com TD3', fontsize=font_size);
    ax5.set_ylabel(r'Inclinação $\theta$ (°)', fontsize=font_size);
    ax5.set_xlabel(r'$t$ (s)', fontsize=font_size);
    ax5.set_xlim([0, max_t_TD3])
    ax5.tick_params(axis='x', labelsize=font_size)
    ax5.tick_params(axis='y', labelsize=font_size)
    ax5.grid(True, which='both')
```

```
    ax6.plot(models_TD3[i].get_t(), models_TD3[i].get_x()*100, linewidth=linewidth)
    ax6.set_title('Testes com TD3', fontsize=font_size);
    ax6.set_ylabel(r'Posição $x$ (cm)', fontsize=font_size);
    ax6.set_xlabel(r'$t$ (s)', fontsize=font_size);
    ax6.set_xlim([0, max_t_TD3])
    ax6.tick_params(axis='x', labelsize=font_size)
    ax6.tick_params(axis='y', labelsize=font_size)
    ax6.grid(True, which='both')
fig.tight_layout()
#plt.savefig('resultado.png', dpi=200)
plt.show()
```