Universidade de Brasília
Faculdade de Tecnologia

Aprendizado por reforço aplicado à categoria VSSS de futebol de robôs

Hiago dos Santos Rabelo

PROJETO FINAL DE CURSO

ENGENHARIA DE CONTROLE E AUTOMAÇÃO

Brasília

2023

Universidade de Brasília
Faculdade de Tecnologia

Aprendizado por reforço aplicado à categoria VSSS de futebol de robôs

Hiago dos Santos Rabelo

Projeto Final de Curso submetido como requisito parcial para obtenção do grau de Engenheiro de Controle e Automação

Orientador: Prof. Dr. Marcus Vinicius Lamar
Coorientadora: Prof. Dr. Carla Cavalcante Koike

Brasília
2023

Rabelo, Hiago dos Santos.
R114a Aprendizado por reforço aplicado à categoria VSSS de futebol de robôs / Hiago dos Santos Rabelo; orientador Marcus Vinicius Lamar; coorientadora Carla Cavalcante Koike. -- Brasília, 2023.
96 p.

Projeto Final de Curso (Engenharia de Controle e Automação) -- Universidade de Brasília, 2023.

1. Redes neurais. 2. Aprendizado por reforço. 3. Futebol de robôs. I. Lamar, Marcus Vinicius, orient. II. Koike, Carla Cavalcante, coorient. III. Título

Universidade de Brasília
Faculdade de Tecnologia

Aprendizado por reforço aplicado à categoria VSSS de futebol de robôs

Hiago dos Santos Rabelo

Projeto Final de Curso submetido como requisito parcial para obtenção do grau de Engenheiro de Controle e Automação

Trabalho aprovado. Brasília, 21 de julho de 2023:

Prof. Dr. Marcus Vinicius Lamar
UnB/IE/CIC
Orientador

Profa. Dra. Carla Maria Chagas e Cavalcante Koike
UnB/IE/CIC
Coorientador

Prof. Dr. Alexandre Ricardo Soares Romariz,
UnB/FT/ENE
Examinador interno

Prof. Dr. Guilherme Novaes Ramos
UnB/IE/CIC
Examinador interno

Brasília
2023

Agradecimentos

Agradeço primeiramente a minha mãe, a senhora Maria Edite dos Santos, que é um exemplo de esforço e perseverança. Primeiramente, ao ter tido que sair de casa com 14 anos de idade para poder ajudar a família no interior do Ceará. E depois, ao ter tido que cuidar sozinha de 3 filhos.

Agradeço também ao Galt Vestibulares, um cursinho voluntário pré-vestibular que ajuda estudantes carentes a ingressarem no ensino superior.

Meus agradecimentos vão também para a equipe UnBall, que me proporcionou várias experiências técnicas e aplicações das várias matérias vistas ao longo da graduação.

Agradeço também ao meu amigo Fábio que me deu conselhos ao longo de toda a graduação, em que mesmo não sendo da área de exatas, me ajudou a ser uma pessoa e um profissional melhor.

Sou grato ao meu amigo Gabriel, mesmo que ele já tenha partido, pois ele me mostrou que é possível vivenciar vários momentos felizes durante a graduação e que a faculdade é muito mais do que apenas uma sala de aula. Com o passar dos anos, podemos esquecer o conteúdo estudado nas disciplinas, mas sempre nos lembraremos dos amigos que fizemos e dos momentos especiais que compartilhamos com eles.

Por fim, gostaria de expressar minha gratidão aos professores Marcus Vinícius Lamar e Carla Cavalcante Koike, que me orientaram durante o desenvolvimento desse trabalho.

Resumo

O aprendizado por reforço é um dos campos de aprendizado de máquina que está ganhando bastante destaque atualmente, especialmente com a utilização de redes neurais e a obtenção de resultados que superam valores obtidos por seres humanos. Devido a isso, há um grande espaço para sua aplicação na robótica. Este trabalho tem como objetivo utilizar técnicas de aprendizado de máquina para obter um agente capaz de realizar gols em uma partida de futebol de robôs.

Será utilizado o algoritmo atual da equipe UnBall como referência para comparação nos resultados numéricos. A UnBall é um projeto de extensão da UnB que compete com foco em futebol de robôs autônomos na categoria *Very Small Size Soccer* (VSSS).

Este trabalho possui foco na utilização dos algoritmos *Deep Deterministic Policy Gradient* (DDPG) e *Twin Delayed DDPG* (TD3). Embora os agentes obtidos por aprendizado por reforço possam não superar o algoritmo atualmente utilizado pela equipe UnBall, os resultados obtidos indicam que são efetivos na realização da tarefa de direcionar a bola até o gol.

Palavras-chave: Redes neurais. Aprendizado por reforço. Futebol de robôs.

Abstract

Reinforcement learning is one of the fields of machine learning that is gaining considerable attention, especially with the use of neural networks and achieving results that outperform those obtained by humans. As a result, there is significant potential for its application in robotics. This work aims to utilize machine learning techniques to obtain an agent capable of scoring goals in a robot soccer game.

The current algorithm of the UnBall team will be used as a reference for comparison in the numerical results. UnBall is an extension project of UnB that competes with a focus on autonomous robot soccer in the Very Small Size Soccer (VSSS) category.

This work will focus on the utilization of Deep Deterministic Policy Gradient (DDPG) and Twin Delayed DDPG (TD3) algorithms. Although the agents obtained through reinforcement learning may not outperform the algorithm currently used by the UnBall team, they are expected to be effective in directing the ball toward the goal.

Keywords: Neural network. Reinforcement learning. Robot soccer.

Lista de ilustrações

Figura 2.1 – Função de ativação sigmoide. 17
Figura 2.2 – Função de ativação tangente hiperbólica (tanh). 18
Figura 2.3 – Interação agente-ambiente em aprendizado por reforço. 19
Figura 2.4 – *Deep Q-Network* vs *Q-Learning*. 23
Figura 2.5 – Problema de alvo móvel presente no algoritmo DQN. 24
Figura 2.6 – Correção do problema de alvo móvel com a utilização de uma segunda rede denominada *target-network*. 25
Figura 2.7 – No algoritmo DDPG a rede *actor* recebe um estado s_t, o que produz uma ação a_t. O par estado-ação s_t, a_t são passados para a rede *critic*, à qual produz o valor Q $\hat{Q}$ que é utilizado como o *loss* da rede *actor* durante o treinamento. 31
Figura 2.8 – Esquecimento catastrófico em agentes de aprendizado por reforço. . . . 32
Figura 2.9 – Categoria VSSS de futebol de robôs. 33
Figura 2.10–Diagrama das partes presentes no futebol de robôs. 34
Figura 2.11–P_g denota a posição do gol adversário, P_r denota a posição do robô e P_b a posição da bola. A informação do ângulo do robô é presente em P_r. . . . 35
Figura 2.12–Atribuições das entidades no futebol de robôs. O goleiro atua na pequena área (PA) para evitar que a bola alcance o gol de sua própria equipe (GP). O zagueiro atua em seu próprio campo, impedindo a progressão da bola em direção ao gol aliado. O atacante tenta levar a bola em direção ao gol adversário (GA). 35
Figura 2.13–Estimação de onde o goleiro deve interceptar a bola. 36
Figura 2.14–Modelagem matemática da estimativa de onde a bola vai entrar no gol. . 37
Figura 2.15–Simulador FIRASim. 38
Figura 2.16–Modelagem matemática da estimativa de onde a bola vai entrar no gol. . 39
Figura 2.17–alvo desejado para o robô que possui a entidade atacante. 40
Figura 2.18–O vetor unitário de $P_b - P_g$ multiplicado com uma constante c pode ser utilizado para estimar a posição desejada K atrás da bola na qual o robô fique alinhado com o gol adversário. 40
Figura 2.19–Dubins. 42
Figura 2.21–O vetor unitário em (x,y), denotado pela cor vermelha, representa o ângulo do campo desejado para o robô na posição x,y e de forma que caso alinhe o ângulo dele com os dos campo o torna possível chegar na bola alinhado com o veto em azul. 42

Figura 2.20–O ponto A é representado pelo robô e o ponto B pela bola com o ângulo de chegada dado em função do vetor relativo de posição da bola até o gol. V_b é a velocidade da bola. 43
Figura 3.22–A projeção do vetor de velocidade do robô V_r sobre o vetor unitário que aponta da posição do robô P_r até a posição da bola P_b indica se o robô está se aproximando ou se afastando da bola. Essa projeção é utilizada na função de recompensa R_t para determinar a pontuação do agente. . . . 48
Figura 3.23–A projeção do vetor de velocidade da bola V_b sobre o vetor unitário que aponta da posição da bola P_b até a posição do gol P_g indica se a bola está se aproximando ou se afastando do gol. Essa projeção é utilizada na função de recompensa R_t para determinar a pontuação. 48
Figura 3.24–Funções de ativação Mish e ReLU. 51
Figura 3.25–Arquitetura da rede usada para definir o *actor* e para definir o *critic*. V e W denotam a velocidade linear a angular, respectivamente. 52
Figura 4.26–Recompensa ao longo do treinamento com suavização usando média móvel com fator 0.95 56
Figura 4.27–Sequência do jogador após a ocorrência de esquecimento catastrófico. Mantém o aprendizado de como ir para a bola, mas fica parado ao lado dela. 57
Figura 4.28–Recompensa ao longo do treinamento com suavização usando média móvel com fator 0.95 59
Figura 4.29–Sequência com a bola em movimento. O robô se desloca para uma posição que possibilita a interceptação da bola e a realização do gol. 60
Figura 4.30–Sequência do jogador chegando atrás da bola de forma a ficar alinhado para “chutar” a bola ao gol adversário. 61
Figura 4.31–Tempo gasto para obter a menor distância da bola ao gol. Violinos com informação da média ao centro. Valores menores indicam melhores resultados. 63
Figura 4.32–Menor distância obtida da bola em relação ao gol. Violinos com informação da média ao centro. Valores menores indicam melhores resultados. 63

Lista de tabelas

Tabela 2.1 – Exemplo de uma *Q-Table* 21
Tabela 4.1 – Distância da bola ao gol adversário. 62

Lista de abreviaturas e siglas

ANN Rede neural artificial . . . 16

DDPG Deep Deterministic Policy Gradient . . . 25

DDQN *Double Deep Q-Network* . . . 24

DQN *Deep Q-Network* . . . 23

MDP Processo de decisão de Markov, do inglês *Markov Decision Process* . . . 18

MLP Multilayer Perceptron . . . 38

MSE Erro Quadrático Médio, do inglês *Mean Square Error* . . . 23

ReLU Unidade linear retificadora, do inglês *Rectified Linear Unit* . . . 51

TD3 Twin Delayed Deep Deterministic Policy Gradient . . . 31

VSSS Very Small Size Soccer . . . 33

Lista de símbolos

Símbolos romanos

$A(s_t)$ Conjunto de ações possíveis no estado s_t . . . 22
a_t Ação em um tempo t . . . 18
G_t Somatório descontado de recompensas esperadas . . . 18
L Função de *loss* . . . 29
P_b Vetor posição da bola com origem no centro do campo . . . 34
P_g Vetor posição do gol com origem no centro do campo . . . 34
P_{rb} Vetor relativo da posição robô até a bola ($P_{rb} = P_b - P_r$) . . . 34
P_r Vetor posição do robô com origem no centro do campo . . . 34
R_t Recompensa obtida pelo agente no tempo t . . . 18
S Conjunto de estados possíveis . . . 18
s_t Estado em um tempo t . . . 18
T Passo de tempo final em um episódio . . . 18
t Passo de tempo . . . 20
V_b Vetor velocidade da bola . . . 36
V_r Vetor velocidade do robô . . . 41
W Velocidade angular do robô . . . 41

Símbolos gregos

α Taxa de aprendizado . . . 20
ϵ Decaimento na exploração (*Epsilon Decay*) . . . 22
γ Taxa de desconto . . . 19
ω Pesos da rede neural . . . 16
$\pi(a \mid s)$ Política do agente . . . 19
σ Função de ativação . . . 17
θ Ângulo no sentido anti-horário entre a horizontal e a frente do robô . . . 41
θ' Ângulo entre a frente do robô e o vetor de posição do robô até bola . . . 40
ξ Ruído inserido a ação durante o treinamento . . . 29

Sumário

1 Introdução 14

2 Revisão Teórica 16
2.1 Aprendizado de máquina 16
2.2 Redes neurais 16
2.3 Aprendizado por reforço 18
2.3.1 *Q-Learning* 21
2.3.2 *Deep Q-Network* 23
2.3.3 *Double Deep Q-Network* 24
2.3.4 *Policy Gradient* 25
2.3.5 *Actor Critic* (AC) 28
2.3.6 *Deep Deterministic Policy Gradient* (DDPG) 28
2.3.7 *Twin Delayed Deep Deterministic* (TD3) 31
2.4 Futebol de robôs 33
2.5 Simulador 37
2.6 Trabalhos relacionados 39
2.7 Conclusão do capítulo 44

3 Proposta 45
3.1 Informações de entrada dos agentes 45
3.2 Interação com o ambiente 46
3.3 Função de recompensa 47
3.4 *Experience Replay* 47
3.5 Arquitetura das redes neurais e hiperparâmetros 49
3.6 Ruído de exploração 53
3.7 Ferramenta *ONNX Runtime* 53
3.8 Proposta de validação dos agentes 53

4 Resultados 55
4.1 Esquecimento catastrófico nos agentes propostos 56
4.2 Utilização de políticas ótimas locais 59
4.3 Análise quantitativa da desempenho dos agentes 62
4.4 Análise qualitativa do desempenho dos agentes 64
4.5 Considerações sobre os resultados obtidos 64

5 Conclusões 65

Referências **66**

Apêndices **72**

Apêndice A Códigos de programação **73**

A.1 Classe de comunicação entre o FiraSIM e o agente 73

A.2 *Reaplay Buffer* responsável por armazenar os estados e ações 80

A.3 Arquiteturas das redes neurais utilizadas na monografia 83

A.4 Arquivo principal responsável por dar início ao treinamento 90

A.5 Arquivo responsável pela etapa de validação dos agentes 94

A.6 Funções secundárias necessárias para a implantação de utilidades ao longo do projeto 95

1 Introdução

Atualmente, a Inteligência Artificial (IA) está presente em diversos campos, como na área da saúde, onde é utilizada na análise de exames para classificação de doenças (KAUR et al., 2022); na segurança de servidores em nuvem, para detecção e defesa contra invasões (BALAMURUGAN et al., 2022); em carros autônomos, visando a redução de acidentes no trânsito (ELALLID et al., 2022); e em chatbots, que possibilitam a comunicação entre humanos e máquinas (CALDARINI; JAF; MCGARRY, 2022).

No contexto de agentes autônomos, o uso de aprendizado de máquina é amplamente conhecido. No entanto, é interessante mencionar o exemplo do chatGPT, um chatbot que também emprega aprendizado por reforço como parte de seu processo de treinamento para ajuste fino de parâmetros (OUYANG et al., 2022).

As aplicações mencionadas destacam a importância do estudo de algoritmos de aprendizado por reforço. Nesse contexto, há uma área específica da robótica que ainda é pouco explorada em relação à aplicação dessas técnicas: a categoria *Very Small Size Soccer* (VSSS), que é uma categoria de futebol de robôs presente na competição internacional LARC (*Latin American Robotics Competition*). No VSSS partida é composta por dois times de 3 robôs, onde cada robô pode ter as dimensões de 75×75 mm. Nessa área, os algoritmos de aprendizado por reforço apresentam um grande potencial devido à variedade de movimentos que os agentes podem aprender, com foco no resultado final de marcar gols.

A utilização de uma função de recompensa desempenha um papel fundamental no treinamento do agente no ambiente por meio de algoritmos de aprendizado por reforço. A proposta de uma função de recompensa específica para essa categoria de futebol de robôs, apresentada por (MARTINS et al., 2021), possibilita a aplicação dessas técnicas nesse contexto.

Este trabalho tem como objetivo treinar um robô para jogar futebol de robôs em um simulador e avaliar a viabilidade de substituir os algoritmos atuais baseados em máquina de estados[1] por um agente obtido por aprendizado por reforço. Um objetivo secundário é implementar um sistema que permita a obtenção das recompensas obtidas pelo agente ao interagir no simulador do ambiente, seguindo a função de recompensa proposta por (MARTINS et al., 2021).

É importante destacar que a equipe vencedora atual na categoria VSSS da Larc 2021 é a RobôCIn, que é o grupo de pesquisa da UFPE. A equipe RobôCIn se destaca por ser a única a utilizar o aprendizado por reforço como substituto completo do bloco de estratégia. Já segunda colocada é a equipe ITAndroids, que representa o time de robótica do ITA,

[1] Os códigos utilizados no trabalho estão disponíveis em: `https://github.com/hiagop22/unball_ia`

utiliza parcialmente o aprendizado por reforço em seu sistema. Esses exemplos demonstram o potencial do aprendizado por reforço em superar os blocos de estratégia baseados em máquinas de estados.

Dado o que foi apresentado, este estudo busca investigar duas hipóteses principais: a primeira é se é viável desenvolver um agente autônomo capaz de marcar gols usando técnicas de aprendizado por reforço que possa se equiparar ao sistema implementado da UnBall criado partir de (KIM et al., 2004); a segunda hipótese é como é possível construir uma rede neural para ser empregada como agente nesse contexto. Além disso, busca-se verificar se essa rede neural proposta é capaz de competir de forma efetiva com algoritmos que utilizam a arquitetura atual baseada em máquina de estados.

No Capítulo 2, são discutidos os conceitos teóricos relacionados ao aprendizado de máquina e seus diferentes campos. Em seguida, é feita uma breve revisão sobre redes neurais e seu funcionamento. Uma análise teórica dos algoritmos de aprendizado de máquina utilizados neste trabalho é apresentada, seguida pela exploração do tema do futebol de robôs. São apresentadas as abordagens atuais para lidar com os desafios dessa modalidade. No final do Capítulo 2, é introduzido o simulador que será utilizado no trabalho, bem como as propostas para desenvolver um agente capaz de marcar gols.

No Capítulo 3, é apresentado embasamento teórico adotado na literatura para construir uma rede neural capaz de atuar como agente no aprendizado por reforço.

No Capítulo 4, são apresentados os resultados obtidos para cada um dos agentes treinados, bem como uma comparação entre esses resultados e o desempenho do algoritmo atualmente utilizado pela equipe UnBall.

Por fim, o Capítulo 5 discute as conclusões alcançadas e sugere possíveis trabalhos futuros para abordar os desafios encontrados durante a elaboração desta monografia.

2 Revisão Teórica

Este capítulo aborda o funcionamento de redes neurais e fornece uma explicação matemática dos algoritmos de aprendizado por reforço, abrangendo desde os mais simples até o TD3 (*Twin Deterministic DDPG*), considerado um dos métodos mais avançados atualmente. Em seguida, é apresentado o contexto do futebol de robôs e são discutidas algumas abordagens utilizadas para resolver os desafios de engenharia encontrados durante a implementação da tomada de decisões do robô.

2.1 Aprendizado de máquina

O aprendizado de máquina é um ramo da inteligência artificial que se concentra no desenvolvimento de algoritmos capazes de aprender a realizar tarefas específicas sem serem programados diretamente para isso (HELM et al., 2020). Existem três tipos principais de tarefas de aprendizado: aprendizado supervisionado, aprendizado não supervisionado e aprendizado por reforço (MURDOCH et al., 2019).

No aprendizado supervisionado, os algoritmos utilizam um conjunto de dados de entrada com rótulos prévios para as saídas correspondentes. O objetivo é encontrar uma função que relacione as entradas aos rótulos de saída (BURKART; HUBER, 2021).

No aprendizado não supervisionado, o objetivo é agrupar os dados. A partir de um conjunto de dados, o algoritmo busca separá-los em subconjuntos, chamados de grupos, de forma que os elementos dentro de cada grupo tenham características semelhantes entre si (USAMA et al., 2019).

O foco deste trabalho está no aprendizado por reforço, onde um agente precisa aprender a se comportar em um ambiente interagindo com ele. Nesse tipo de aprendizado, o agente recebe recompensas ou penalidades com base em suas ações, buscando aprender a tomar decisões que maximizem as recompensas ao longo do tempo (SAHU; MOKHADE; BOKDE, 2023).

2.2 Redes neurais

Dentro do campo de aprendizado de máquina, o *deep learning* desempenha um papel importante, com o uso generalizado de algoritmos de redes neurais. Uma rede neural artificial (ANN, *Artificial Neural Network*) é composta por neurônios artificiais interconectados (ABIODUN et al., 2019). Sua função é criar uma função matemática que relacione um conjunto de entradas X a um conjunto de saídas Y.

Cada neurônio artificial em uma rede neural modela o comportamento de um neurônio biológico (PATURI; CHERUKU; REDDY, 2022). Ele recebe um conjunto de n entradas x_j (onde $j = 0,1,2,...,n$), que são multiplicadas pelos pesos correspondentes w_j e somadas a um viés (bias) θ. O resultado dessa operação é denominada Potencial de Ativação (u), sendo expresso por

$$u = \sum_{j=0}^{n} w_j \cdot x_j + \theta. \tag{2.1}$$

O Potencial de Ativação (u) é então passado por uma Função de Ativação, denotada por ϕ, que determina a amplitude da saída y do neurônio (JUAN; VALDECANTOS, 2022).

Uma aplicação comum das funções de ativação é restringir o intervalo de saída, por exemplo, para os valores entre 0 e 1, ou entre -1 e 1. O primeiro intervalo pode ser obtido utilizando a função sigmoide (DUBEY; SINGH; CHAUDHURI, 2022), definida por

$$\phi(u) = \frac{1}{1 + e^{-u}}, \tag{2.2}$$

e ilustrada na Figura 2.1. Já o segundo intervalo pode ser obtido utilizando a função tangente hiperbólica (DUBEY; SINGH; CHAUDHURI, 2022), $\phi(u) = \tanh(u)$, como mostrado na Figura 2.2. O presente trabalho utiliza a arquitetura perceptron multicamadas como implementação de redes neurais, o que é amplamente utilizado na comunidade de aprendizagem de máquina (RIJSDIJK et al., 2021).

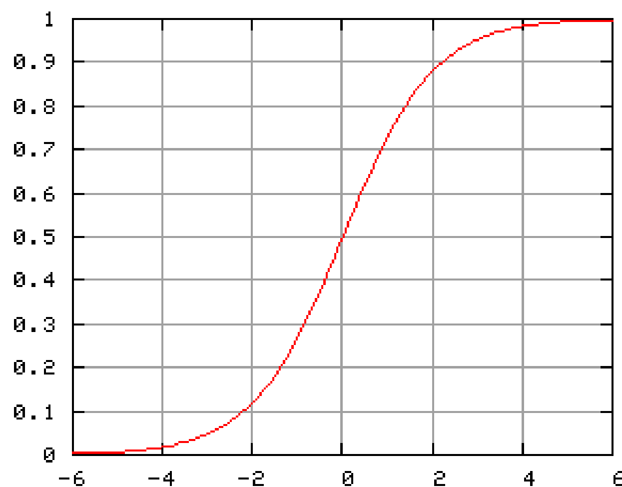

Figura 2.1 – Função de ativação sigmoide.
Fonte: Wikipédia (2021)

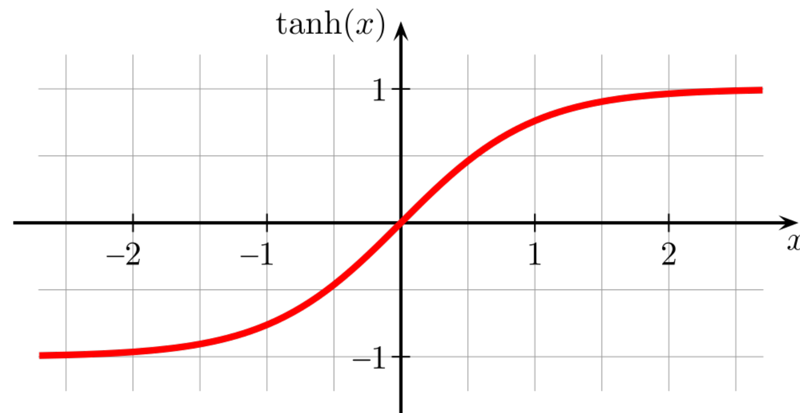

Figura 2.2 – Função de ativação tangente hiperbólica (tanh).
Fonte: Wikipédia (2023)

2.3 Aprendizado por reforço

O aprendizado por reforço é um problema no qual um agente é treinado por meio da interação com ambiente. Essa interação ocorre em passos de tempos t. Tem-se que $t \in E$ e $E = \{t : 0 < t < T;\ \text{onde}\ t,T \in Z_+\}$, sendo T o passo terminal e E o conjunto de passos até o passo terminal, denominado de episódio (SUTTON; BARTO, 2014).

O problema de interação entre o agente e o ambiente é modelado matematicamente como um processo de decisão de Markov (MDP) (SUTTON; BARTO, 2014). Um MDP é uma tupla composta por variáveis de ação, recompensa, probabilidade de o agente executar uma determinada ação e estado.

Define-se A como o conjunto de ações possíveis do agente, enquanto o conjunto de estados possíveis denotado por S. Sendo que, em um tempo t agente recebe como entrada o estado s_t e uma recompensa R_t, e fornece de saída uma ação a_t relacionada ao estado s_t recebido (SUTTON; BARTO, 2014). O ambiente, por sua vez, recebe a ação a_t como entrada e retorna no tempo $t+1$ uma recompensa R_{t+1} relacionada ao novo estado s_{t+1}. Essas transições de estado e recompensa são determinadas pela ação a_t, conforme ilustrado na Figura 2.3.

Podemos definir $\hat{G}_t$ como o somatório das recompensas esperadas, ou seja, $\hat{G}_t = R_{t+1} + R_{t+2} + ...R_{t+T}$, onde T é o passo final (SUTTON; BARTO, 2014). Então $\hat{G}_t$ pode ser reescrito como

$$\hat{G}_t = \sum_{k=0}^{T-1} R_{t+k+1}. \tag{2.3}$$

É importante ressaltar que a recompensa R_t não é levada em consideração durante a otimização, uma vez que o estado inicial s_0 é gerado aleatoriamente (SUTTON; BARTO,

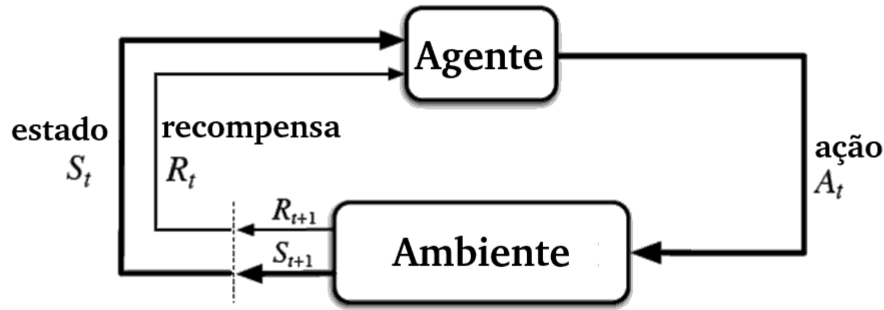

Figura 2.3 – Interação agente-ambiente em aprendizado por reforço.
Adaptado de: Sutton e Barto (2014, p. 54)

2014).

Assim, um objetivo inicial do aprendizado por reforço pode ser definido como a obtenção de um agente que realize ações a_t que maximizem $\hat{G}_t$. Porém $\hat{G}_t$ não é a melhor função a ser usada para a otimização, uma vez que todas as recompensas possuem o mesmo peso (SUTTON; BARTO, 2014). É necessário atribuir pesos diferentes às recompensas futuras estimadas, pois uma recompensa estimada para um tempo $t + m$ é mais provável de ser obtida do que uma recompensa em um tempo $t + n$, com $m < n$ e $\{m, n\} \in \mathbb{Z}_+$.

Para incorporar esse conceito, introduz-se um fator de desconto γ nas recompensas esperadas da Equação 2.3, com $0 \leq \gamma \leq 1$. Isso resulta em uma função G_t chamada de somatório descontado de recompensas esperadas (SUTTON; BARTO, 2014), definida por

$$G_t = \sum_{k=0}^{T-1} \gamma^k R_{t+k+1} \tag{2.4}$$

Uma função de política π é definida como uma função que mapeia o domínio de estados S para o domínio de ações A. Se a política for determinística, ela é expressa como $\pi(s) = s \to a$, relacionando diretamente um estado s_t a uma ação a_t. A literatura (SUTTON; BARTO, 2014) também utiliza a notação $\mu(s)$ para denotar $\pi(s)$ quando se trata de uma política determinística. Por outro lado, se a política for estocástica, ela é definida como $\pi(a|s) = P[a|s]$, fornecendo a probabilidade de selecionar a ação a_t dado o estado s_t, onde a se refere a a_t e s se refere a s_t (SUTTON; BARTO, 2014).

A função de valor $V_\pi(S)$ é obtida através do somatório descontado de recompensas esperadas G_t (SUTTON; BARTO, 2014). Essa função representa a esperança de G_t dado o estado s_t e é denotada como $V^\pi(s) = E_\pi[G_t|s = s_t]$. A função de valor é uma medida do desempenho do agente em um estado, seguindo a política π, e é definida por

$$V^\pi(s) = E_\pi\left[\sum_{k=0}^{T-1} \gamma^k R_{t+k+1} \middle| s = s_t\right]. \tag{2.5}$$

Uma função de valor ótima (SUTTON; BARTO, 2014), denotada como V_*, é a função de valor máximo que pode ser alcançada considerando possíveis políticas V^π, e é definida por

$$V_*(s) = max_\pi V_\pi(s). \tag{2.6}$$

A função de ação-valor (SUTTON; BARTO, 2014), denotada por $Q^\pi(s,a)$ é mostrada na Equação 2.7. A função $Q^\pi(s,a)$ também avalia a qualidade de um agente em um estado s_t ao seguir uma política π, porém considerando especificamente a realização da ação a_t no estado s_t.

$$\begin{aligned} Q^\pi(s,a) &= E_\pi[G_t | s = s_t, a = a_t] \\ &= E_\pi\left[\sum_{k=0}^{T-1} \gamma^k R_{t+k+1} \middle| s = s_t, a = a_t\right] \end{aligned} \tag{2.7}$$

De forma similar à Equação 2.6, uma função de valor ótima Q_* é a função com maior valor que pode ser obtida considerando todas as possibilidades de $Q^\pi(s,a)$ para todas as políticas possíveis (SUTTON; BARTO, 2014). Essa função é denotada como:

$$Q_*(s,a) = max_\pi Q_\pi(s,a) \tag{2.8}$$

Caso seja possível alcançar o estado terminal s_t, a tarefa é dita episódica (SUTTON; BARTO, 2014). Para atualizar a função de valor, pode-se esperar o agente realizar uma sequência de ações no ambiente, armazenando as tuplas de informações (R_t, s_t, a_t, s_{t+1}) para cada passo de tempo até que seja alcançado o estado terminal s_T. Em seguida, é possível obter o valor da função G_t para cada passo de tempo t. Obtidos os valores de G_t, o agente é treinado. Essa abordagem é conhecida como Monte Carlo (SUTTON; BARTO, 2014) e utiliza o parâmetro α como a taxa de aprendizado, onde $0 < \alpha \leq 1$. A atualização da função de valor $V(s_t)$ é dada por:

$$V(s_t) \leftarrow V(s_t) + \alpha[G_t - V(s_t)] \tag{2.9}$$

Caso o passo terminal $T \to \infty$, então a tarefa ao qual o agente está sendo aplicado é denominada de tarefa contínua (SUTTON; BARTO, 2014). No treinamento de um agente nessa tarefa, não é possível obter o valor de G_t para todos os passos, uma vez que não há um passo terminal T definido. Portanto, é necessário treinar o agente enquanto ele interage com o ambiente, e para isso, utiliza-se o aprendizado de diferença temporal (TD) a cada passo de tempo.

A atualização da função de valor usando o aprendizado TD é definida como

$$V(s_t) \leftarrow V(s_t) + \alpha[R_t + \gamma V(s_{t+1}) - V(s_t)]. \quad (2.10)$$

Na Equação 2.10, a expressão $R_t + \gamma V(s_{t+1})$ é denominada TD alvo, enquanto $R_t + \gamma V(s_{t+1}) - V(s_t)$ é denominada TD *error* (SUTTON; BARTO, 2014).

Existem duas abordagens para se resolver o problema de aprendizado por reforço: a abordagem *model-free* e a abordagem *model-based*. Um algoritmo é considerado *model-based* se, após o treinamento, o agente é capaz de prever o próximo estado, a próxima recompensa e tomar ações baseadas nessas predições. Caso contrário, é considerado *model-free* (KAISER et al., 2019).

Os algoritmos *model-free* podem ser baseados em valores, os quais tentam otimizar a função de valor $V_\pi(s)$, ou podem ser baseados em políticas, os quais tentam otimizar a política $\pi(s)$ sem a necessidade de se usar uma função de valor. No presente trabalho, serão abordados apenas com algoritmos *model-free* (SUTTON; BARTO, 2014).

2.3.1 *Q-Learning*

O *Q-Learning*, proposto por (WATKINS; DAYAN, 1992), também é conhecido como *Q-Table*, é um algoritmo que tem como objetivo preencher uma tabela, onde as linhas representam os estados pertencentes à S, e relacioná-los com as ações pertencentes a A através de um valor denominado valor Q. O valor Q é um número que indica a qualidade de uma ação a_t no estado s_t. Um exemplo de valor Q é mostrada na Tabela 2.1, e quanto maior o valor de valor Q, melhor é a ação no estado correspondente.

Tabela 2.1 – Exemplo de uma *Q-Table*

Estado	Ação1	Ação2	Ação2
S_1	-2.5	1.0	5.6
S_2	3.3	1.5	2.2
S_3	0.2	-1.0	-0.2
S_4	2.1	3.3	4.1

Fonte: Produzido pelos autor.

Nota: A tabela foi inicializada com valores arbitrários de valor Q a título de ilustração.

Para encontrar o valor de cada valor Q relacionado às ações $a_t \in A$ nos estado $s_t \in S$, utiliza-se a Equação de Bellman (WATKINS; DAYAN, 1992) para um ambiente determinístico descrita por

$$Q(s_t,a_t) = R(s_t,a_t) + \gamma max_a Q(s_{t+1},a). \tag{2.11}$$

A Equação de Bellman (Equação 2.11) descreve como calcular o valor atual do valor Q para uma determinada ação a_t em um estado s_t. Seguindo essa equação, o valor Q é determinado somando a recompensa imediata $R(s_t,a_t)$ com o valor máximo do valor Q para todas as ações possíveis no próximo estado s_{t+1}. Especificamente, a ação que maximiza o valor Q é selecionada entre todas as opções disponíveis. Antes de ser somado, o valor do próximo estado é multiplicado pelo fator de desconto γ. Essa equação permite atualizar o valor Q iterativamente e leva em consideração tanto a recompensa imediata quanto as recompensas futuras ponderadas pelo fator de desconto.

Para atualizar a *Q-Table*, inicializa-se a tabela com valores arbitrários e, em seguida, utiliza-se um processo iterativo com base na Equação do de subida por gradiente dada por

$$\begin{aligned} Q_{novo}(s_t,a_t) &= Q(s_t, a_t) + \alpha \Delta Q(s_t,a_t) \\ &= Q(s_t,a_t) + \alpha(R(s_t,a_t) + \gamma max_a Q(s_{t+1},a) - Q(s_t,a_t)). \end{aligned} \tag{2.12}$$

Nesse processo, o novo valor Q para um par estado-ação (S, A) é calculado somando-se o valor Q atual com a diferença $\Delta Q(S,A)$, multiplicada por uma taxa de aprendizado α (onde $0 < \alpha \leq 1$). A diferença $\Delta Q(S,A)$ é calculada como a diferença entre o valor Q obtido na interação atual com o ambiente e o valor Q anterior armazenado na tabela. Esse processo de atualização iterativo permite que a *Q-Table* seja gradualmente ajustada com base nas experiências do agente durante a interação com o ambiente.

Para preencher a tabela de valor Q, inicialmente com valores arbitrários, o agente adota uma estratégia de exploração, que prioriza explorar o ambiente em vez de escolher a ação com o maior valor Q (WATKINS; DAYAN, 1992). Com o tempo, conforme o agente adquire mais experiência, é benéfico que ele comece a utilizar as informações contidas na tabela para guiar suas ações, o que é conhecido como estratégia de explotação. Para equilibrar essas abordagens, é comum utilizar uma taxa de decaimento da exploração, denominada *epsilon decay*, representada por ϵ. Essa taxa diminui gradualmente ao longo do treinamento, permitindo que o agente inicialmente explore amplamente o ambiente e, posteriormente, faça mais uso das informações armazenadas na tabela.

Através da utilização da Equação 2.12, é possível que os valores na tabela de valores Q convirjam para os valores ótimos $Q_*(s,a)$. No entanto, uma desvantagem do algoritmo *Q-Learning* é a necessidade de criar uma tabela que contenha uma linha para cada possível estado, o que se torna inviável em ambientes com centenas ou milhares de estados possíveis.

2.3.2 *Deep Q-Network*

A abordagem discutida na subseção 2.3.1 é impraticável em ambientes complexos com centenas ou milhões de estados possíveis. Para enfrentar esse desafio, o trabalho (MNIH et al., 2013) propõe o *Deep Q-Network* (DQN). Em vez de usar uma tabela para relacionar cada tupla (s_t, a_t) a um valor Q, o DQN utiliza uma rede neural que mapeia um estado s_t para n saídas da rede, representando as ações possíveis. Isso é ilustrado na Figura 2.4.

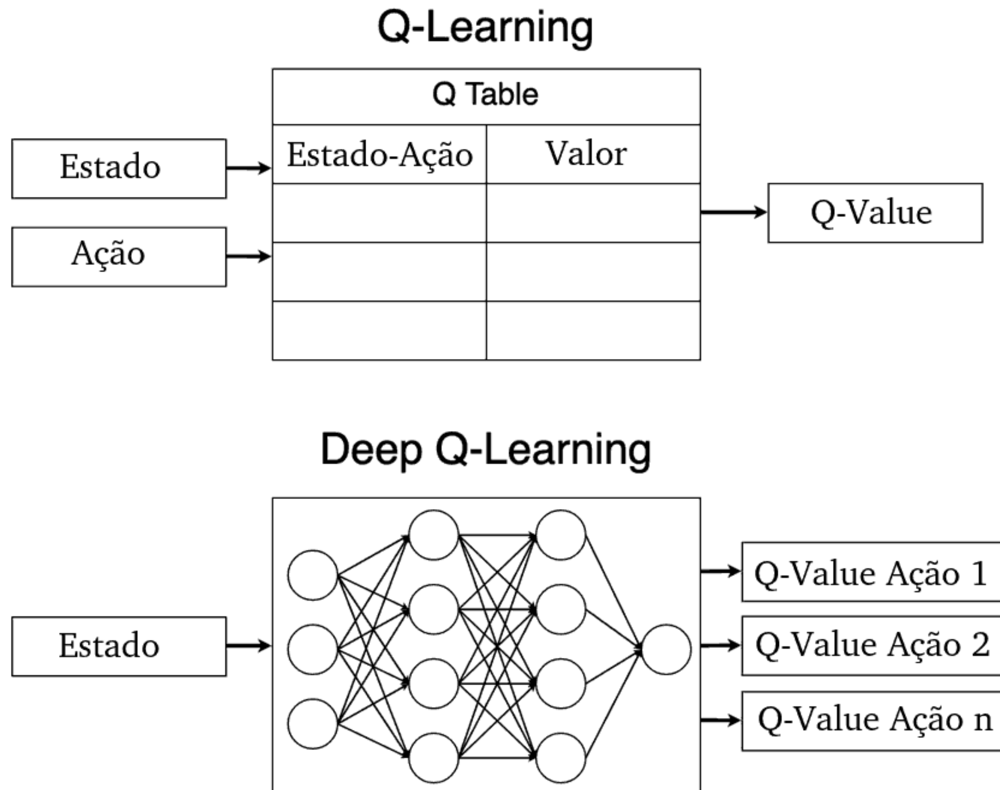

Figura 2.4 – *Deep Q-Network* vs *Q-Learning*.
Adaptado de: Ullo (2021)

A fim de atualizar os pesos da rede neural mencionada na Figura 2.4, é necessário definir uma função de custo L, que quantifica a discrepância entre a saída da rede e a saída desejada (MNIH et al., 2013). Essa função de custo é calculada usando o erro quadrático médio (MSE) entre o valor Q atual $Q(s_t, a_t; w)$ e o valor Q esperado $(R(s_t, a_t) + \gamma \max_a Q(s_{t+1}, a; w))$. A função de custo é determinada por

$$L = \frac{(Q(s_t, a_t; w) - (R(s_t, a_t) + \gamma max_a Q(s_{t+1}, a; w)))^2}{2}. \tag{2.13}$$

Em particular, $Q(s_t, a_t; w)$ representa a saída da rede neural para um conjunto de pesos w, $R(s_t)$ denota as recompensas obtidas no estado s_t e $Q(s_{t+1}, a_{t+1}; w)$ é definido como *Q-target*. Consequentemente, a atualização dos pesos da rede ocorre conforme

$$w_{novo} = w + \alpha(R(s_t, a_t) + \gamma Q_w(s_{t+1}, a_{t+1}) - Q_w(s_t, s_t))\nabla_w Q_w(s_t, a_t) \tag{2.14}$$

onde w_{novo} representa os pesos atualizados, w são os pesos atuais, α é a taxa de aprendizado, $R(s_t, a_t)$ é a recompensa obtida no estado s_t com a ação a_t, γ é o fator de desconto, $Q_w(s_{t+1}, a_{t+1})$ é o valor Q esperado para o próximo estado s_{t+1} e a ação a_{t+1}, $Q_w(s_t, s_t)$ é o valor Q atual para o estado s_t, e $\nabla_w Q_w(s_t, a_t)$ é o gradiente da rede neural em relação aos pesos w no estado s_t e ação a_t.

2.3.3 *Double Deep Q-Network*

O algoritmo *Deep Q-Network* (DQN) proposto por (MNIH et al., 2013), apresenta uma desvantagem em relação à saída da rede neural. O valor Q estimado $Q(s_t, a_t; w)$ tenta se aproximar do *Q-target* $R(s_t, a_t) + \gamma \max_a Q(s_{t+1}, a; w)$, que também depende dos pesos w da rede. Esse problema é ilustrado na Figura 2.5, onde conforme a rede tenta se aproximar do alvo, o alvo também muda devido à atualização dos pesos w, resultando em uma grande oscilação durante o treinamento e podendo levar à divergência.

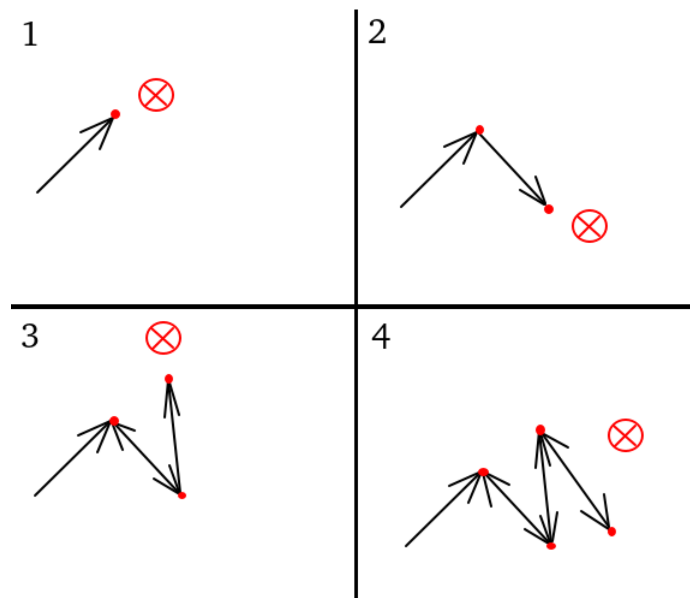

Figura 2.5 – Problema de alvo móvel presente no algoritmo DQN.
Adaptado de: Morales (2020)

Em contraste, o algoritmo *Q-Learning* não sofre com esse problema, uma vez que o valor $\max_a Q(s_{t+1}, a)$ permanece inalterado durante a atualização de $Q(s_t, a_t)$.

Com o objetivo de obter *Q-targets* estáveis, como mostrado na Figura 2.6, o trabalho (VAN HASSELT; GUEZ; SILVER, 2016) propõe o *Double Deep Q-Network* (DDQN).

No método DDQN, é utilizada uma segunda rede neural, chamada de *rede-target*, com pesos w_{targ}. A *rede-target* é inicializada com os mesmos pesos w da rede principal (VAN

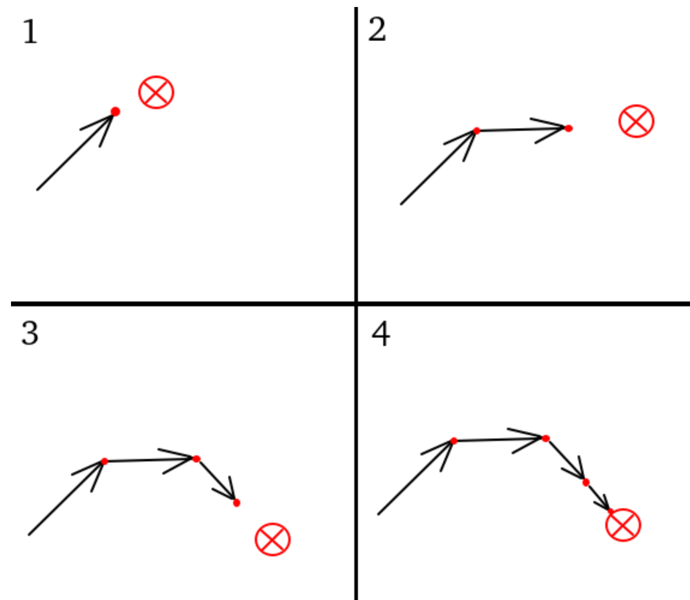

Figura 2.6 – Correção do problema de alvo móvel com a utilização de uma segunda rede denominada *target-network*.
Adaptado de: Morales (2020)

HASSELT; GUEZ; SILVER, 2016). A cada C passos ocorre uma atualização dos pesos w_{targ} por meio do método denominado *soft-update* expresso por

$$w_{targ} = \tau w + (1 - \tau) w_{targ}. \tag{2.15}$$

Com a introdução da *rede-target* de pesos w_{targ}, a função de custo para o treinamento da rede principal de pesos w é definida por

$$Custo = MSE(Q(s_t, a_t; w), (R(s_t, a_t) + \gamma max_a Q(s_{t+1}, a; w_{targ}))). \tag{2.16}$$

Essa equação representa o cálculo do erro quadrático médio (MSE) entre o valor Q atual $Q(s_t, a_t; w)$ e o valor Q esperado $(R(s_t, a_t) + \gamma \max_a Q(s_{t+1}, a; w_{targ}))$.

2.3.4 *Policy Gradient*

O algoritmo DDQN trouxe melhorias em relação ao DQN, mas ainda enfrenta limitações ao lidar apenas com ações discretas. No entanto, muitas aplicações do mundo real envolvem ações contínuas, como a velocidade angular desejada de um robô. Para lidar com esse cenário, o trabalho (SUTTON; MCALLESTER et al., 1999) propõe a otimização da função de política $\pi(s|a)$. Essa abordagem tem a vantagem de não exigir o cálculo do valor Q para decidir qual ação tomar, permitindo a realização de ações exploratórias no ambiente sem a necessidade de definições explícitas de exploração e explotação. Além disso, como

a função de política é estocástica, é possível gerar ações contínuas sem a necessidade de discretizá-las.

O parâmetro τ é definido como uma trajetória $(s_0, a_0, s_1, a_1, ..., s_T, a_T)$ durante o treinamento, que vai desde o tempo inicial $t = 0$ até o estado tempo terminal do episódio $t = T$.

Deste modo, podemos definir a política $\pi_w(\tau)$ como a probabilidade de eventos ocorrerem em uma trajetória τ ao seguir os parâmetros w da rede (SUTTON; MCALLESTER et al., 1999). Essa probabilidade é composta por três elementos: $p_w(S_0)$, que representa a probabilidade de chegar ao estado inicial S_0; $\pi_w(a_t|s_t)$, que representa a probabilidade de escolher a ação a_t dado o estado s_t; e $p(s_{t+1}|s_t, a_t)$, que representa a probabilidade de fazer a transição para o estado s_{t+1} após executar a ação a_t no estado s_t. Essa probabilidade pode ser expressa da seguinte forma:

$$\begin{aligned} \pi_w(\tau) &= p_w(S_0, A_0, S_1, A_1, ..., s_t, a_t) \\ &= p_w(S_0) \prod_{t=0}^{T} \pi_w(a_t|s_t) p(s_{t+1}|s_t, a_t) \end{aligned} \tag{2.17}$$

A função de score $J(w)$ avalia a qualidade de uma política π desde o estado inicial s_0 até o estado terminal s_T utilizando os pesos w da rede, sendo definida como

$$\begin{aligned} J(w) &= E_\pi[G_0(\tau)] = E_\pi[R(\tau)] \\ &= E_\pi[R_1 + \gamma R_2 + \gamma^2 R_3 ...] \end{aligned} \tag{2.18}$$

A função de score utiliza o valor $G_0(\tau)$, que representa o somatório descontado das recompensas esperadas ao seguir uma trajetória τ (SUTTON; MCALLESTER et al., 1999). É importante ressaltar que na literatura, $G_0(\tau)$ também é conhecido como $R(\tau)$. A função de score é definida como:

$$J(w) = E_\pi[R(\tau)] \tag{2.19}$$

Aplicando a Equação 2.18 no algoritmo de gradiente ascendente da Equação 2.20, tenta-se obter os pesos w da rede que maximizam $J(w)$.

$$w_{novo} = w + \alpha \nabla_w J(w) \tag{2.20}$$

O fator $\nabla J(w)$ é expandido para

$$\begin{aligned}\nabla_w J(w) &= \nabla_w E_\pi[R(\tau)] \\ &= \nabla_w \int \pi_w(\tau)R(\tau)d\tau \\ &= \int \nabla_w \pi_w(\tau)R(\tau)d\tau.\end{aligned} \tag{2.21}$$

Utilizando a propriedade da derivada do *log*, mostrada na Equação 2.22, o fator $\nabla_w \pi_w(\tau)$ da Equação 2.21 é substituído por $\pi_w(\tau)\nabla_w log\pi_w(\tau)$ e resultando na Equação 2.23.

$$\pi_w(\tau)\nabla_w log\pi_w(\tau) = \pi_w(\tau)\frac{\nabla_w \pi_w(\tau)}{\pi_w(\tau)} = \nabla_w \pi_w(\tau) \tag{2.22}$$

$$\begin{aligned}\nabla_w J(w) &= \int \pi_w \nabla_w log\pi_w(\tau)R(\tau)d\tau \\ &= E_\pi[\nabla_w log\pi_w(\tau)R(\tau)]\end{aligned} \tag{2.23}$$

Dado que a probabilidade do estado inicial e terminal ocorrerem é sempre 1 em uma tarefa episódica e aplicando a Equação 2.17 na Equação 2.23, obtém-se

$$\begin{aligned}\nabla_w J(w) &= E_\pi[\nabla_w log[p_w(S_0)\prod_{t=0}^{T}\pi_w(a_t|s_t)p(s_{t+1}|s_t,a_t)]R(\tau)] \\ &= E_\pi[\nabla_w[log p_w(S_0) + \sum_{t=0}^{T} log\pi_w(a_t|s_t) + log p(s_{t+1}|s_t,a_t)]R(\tau)] \\ &= E_\pi[\nabla_w[\sum_{t=0}^{T} log\pi_w(a_t|s_t)]R(\tau)] \\ &= E_\pi[\nabla_w[\sum_{t=0}^{T} log\pi_w(a_t|s_t)]G_{t=0}(\tau)].\end{aligned} \tag{2.24}$$

A partir da Equação 2.24, podemos utilizar amostras de ações a e estados s de uma trajetória τ definida por uma política π_w para treinar a rede w usando o cálculo de G_t ao final de cada episódio (SUTTON; MCALLESTER et al., 1999). A atualização dos pesos w da rede pode ser feita pela equação

$$\Delta_w = \alpha\nabla_w[log\pi_w(a_t|s_t)]G_{t=0}(\tau). \tag{2.25}$$

2.3.5 *Actor Critic* (AC)

O algoritmo de *Policy Gradient* utiliza o valor $G_{t=0}(\tau)$ como um indicador para determinar se as ações tomadas em uma trajetória τ são boas ou ruins com base no sinal de $G_{t=0}(\tau)$. Isso representa um avanço nos algoritmos de política. No entanto, surge um problema: ações intermediárias ruins ao longo da trajetória podem ser erroneamente consideradas boas, pois apenas o valor $G_t(\tau)$ em $t = 0$ é usado para avaliar todas as ações como boas ou ruins. Como resultado, todas as ações na trajetória τ são computadas como boas ou ruins em conjunto.

Comparativamente, políticas baseadas apenas em valores não são eficientes quando se lida com um grande número de ações. Para abordar os desafios enfrentados pelos algoritmos baseados exclusivamente em política ou em valor, o trabalho (GRONDMAN et al., 2012) propõe uma a utilização de redes neurais com a abordagem chamada de *Actor Critic*. Nesse método, busca-se combinar elementos dos dois enfoques. O componente *actor* é responsável por executar as ações e é baseado em uma política, enquanto o componente *critic* avalia se as ações realizadas pelo *actor* são boas ou ruins e é baseado em valores. Dessa forma, o *Actor Critic* aproveita as vantagens de ambos os paradigmas, buscando obter um desempenho superior.

O algoritmo *Actor Critic* utiliza a Equação 2.25 para atualizar os pesos w da rede *actor*, porém substituindo o termo $G_{t=0}(\tau)$ por $\hat{Q}y(st, a_t)$. O valor $\hat{Q}y(st, a_t)$ é uma estimativa do valor Q fornecida pela rede *critic* com pesos y conforme

$$w_{novo} = w + \alpha \nabla_w [log\pi_w(a_t|s_t)] \hat{Q}_y(s_t, a_t) \tag{2.26}$$

e

$$y_{novo} = y + \beta(R(s_t, a_t) + \gamma \hat{Q}_y(s_{t+1}, a_{t+1}) - \hat{Q}_y(s_t, s_t)) \nabla_y \hat{Q}_y(s_t, a_t). \tag{2.27}$$

As redes *actor* e *critic* possuem taxas de aprendizado diferentes, denotados por α e β, respectivamente (GRONDMAN et al., 2012).

2.3.6 *Deep Deterministic Policy Gradient* (DDPG)

O algoritmo QAC utiliza uma política estocástica π, o que significa que é necessário gerar uma distribuição de probabilidade para cada par estado-ação s_t, a_t. Embora isso seja comum, a abordagem estocástica pode ser complexa e não é adequada para casos em que desejamos encontrar uma ação determinística ótima para um dado estado. Para abordar esse desafio, o trabalho (LILLICRAP et al., 2015) introduz o algoritmo *Deep Deterministic Policy Gradient* (DDPG).

O DDPG combina a abordagem *actor-critic* com a ideia determinística apresentada em (SILVER et al., 2014). Em vez de uma política estocástica, o DDPG visa encontrar uma política determinística $\mu = \pi(s)$ que sempre gera a melhor ação a_t para um dado estado s_t. Ainda existe a necessidade de se otimizar uma função valor $J(w)$ análoga à apresentada na Equação 2.18. A função valor determinística é definida como $J(w) = E_\mu[Q(s_t, \mu(s_t))]$, onde o desenvolvimento de $\nabla_w J(w)$ através regra da cadeia é dada por

$$\begin{aligned} \nabla_{w^\mu} J(w) &= E_\mu[\nabla_{w^\mu} Q(s_t, \mu(s_t))] \\ &= E_\mu[\nabla_\mu Q(s_t, \mu(s_t)) \nabla_{w^\mu} \mu(s_t)]. \end{aligned} \tag{2.28}$$

Para atualizar os pesos da rede *actor* e *critic* no algoritmo *Actor Critic*, é necessário utilizar o valor estimado $\hat{Q}(s_t, a_t)$ em vez do valor real $Q(s_t, a_t)$. Aqui, w representa os pesos da rede *actor*, enquanto y representa os pesos da rede *critic*. As equações de atualização dos pesos podem ser expressas por

$$w_{novo} = w + \alpha \nabla_\mu \hat{Q}(s_t, \mu(s_t)) \nabla_{w^\mu} \mu(s_t) \tag{2.29}$$

e

$$y_{novo} = y + \beta(R(s_t, a_t) + \gamma \hat{Q}_y(s_{t+1}, a_{t+1}) - \hat{Q}_y(s_t, s_t)) \nabla_y \hat{Q}_y(s_t, a_t). \tag{2.30}$$

Diferentemente da abordagem estocástica do algoritmo *actor-critic*, o *DDPG* (*Deep Deterministic Policy Gradient*) adota uma abordagem diferente. Enquanto o *actor-critic* estocástico recebe apenas o estado atual s_t como entrada para a rede *critic*, o *DDPG* recebe tanto o estado atual s_t quanto a ação tomada a_t como entrada para a rede *critic* (LILLICRAP et al., 2015).

Quando se utilizam políticas determinísticas, surge a questão de sempre selecionar a mesma ação a_t para um dado estado s_t, o que resulta na falta de exploração do agente no ambiente durante o treinamento. Para lidar com esse problema, é comum adicionar um ruído ξ à ação selecionada. Esse ruído pode ser uma distribuição normal $N(0{,}1)$ ou um processo estocástico Ornstein-Uhlenbeck, que também segue uma distribuição normal, mas com o próximo valor dependendo do valor anterior. Dessa forma, a ação a_t usada durante o treinamento é definida como $a_t = \pi(s_t) + \xi$. Isso permite uma exploração mais efetiva do ambiente pelo agente durante o processo de aprendizado (LILLICRAP et al., 2015).

Para se atualizar a rede *critic*, utiliza-se a Equação 2.30. O *loss* da rede *critic* é calculado utilizando $h_t = R_t + \gamma \hat{Q}(s_{t+1}, a_{t+1})$ em

$$L_{critic} = \frac{1}{N}\sum_{t=0}^{T}(h_t - \hat{Q}(s_t, a_t|w))^2. \tag{2.31}$$

Tanto a Equação 2.29 e a Equação 2.30 quanto a Equação 2.31, apresentam o problema conhecido como alvo móvel, discutido na subseção 2.3.2. Visando resolver isso, é utilizada uma abordagem semelhante a proposta inicialmente por (VAN HASSELT; GUEZ; SILVER, 2016) e posteriormente adotada por (LILLICRAP et al., 2015). Nessa abordagem, são criadas redes adicionais chamadas de *actor target* com pesos w_{targ} e *critic target* com pesos y_{targ}, que são cópias das redes *actor* e *critic* originais, respectivamente. As redes alvos são atualizadas de forma gradual por meio da técnica denominada *soft-update*, como mostrado na Equação 2.15, resultando em

$$w_{targ} = \tau w + (1-\tau)w_{targ} \tag{2.32}$$

e

$$y_{targ} = \tau y + (1-\tau)y_{targ}. \tag{2.33}$$

Essa estratégia de atualização suave permite que as redes alvos acompanhem gradualmente as redes originais, contribuindo para a estabilidade do processo de treinamento.

Assim, a Equação 2.29 e a Equação 2.30 são reescritas como

$$w_{novo} = w + \alpha\nabla_{\mu}\hat{Q}(s_t, \mu(s_t))\nabla_{w^{\mu}}\mu(s_t) \tag{2.34}$$

e

$$y_{novo} = y + \beta(R(s_t, a_t) + \gamma\hat{Q}_{ytarg}(s_{t+1},a_{t+1}) - \hat{Q}_y(s_t, s_t))\nabla_y\hat{Q}_y(s_t, a_t). \tag{2.35}$$

A atualização dos pesos y da rede *critic*, conforme apresentado na Equação 2.30 e na Equação 2.35, utiliza $\mu_{targ}(s_{t+1}) = a_{t+1}$ e tem sua função de *loss* reescrita como

$$h_t = r_t + \gamma\hat{Q}_{targ}(s_{t+1}, \mu_{targ}(s_{t+1})) \tag{2.36}$$

e

$$L_{critic} = \frac{1}{N}\sum_{t=0}^{T}(h_t - \hat{Q}(s_t, a_t))^2. \quad (2.37)$$

De forma visual, o algoritmo DDPG pode ser pode ser resumido pela Figura 2.7.

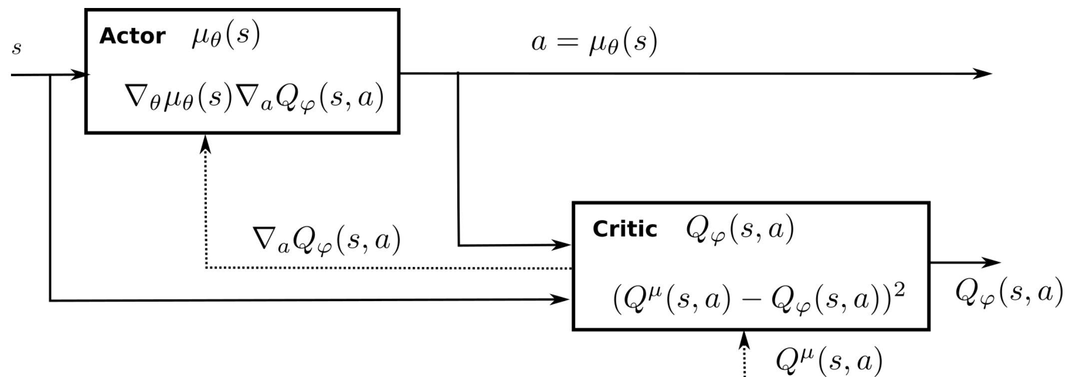

Figura 2.7 – No algoritmo DDPG a rede *actor* recebe um estado s_t, o que produz uma ação a_t. O par estado-ação s_t, a_t são passados para a rede *critic*, à qual produz o valor Q $\hat{Q}$ que é utilizado como o *loss* da rede *actor* durante o treinamento.

Fonte: Gominsu (2008)

2.3.7 *Twin Delayed Deep Deterministic* (TD3)

Apesar de o *Deep Deterministic Policy Gradients* apresentar resultados notáveis no contexto de aprendizado por reforço, ainda apresenta o problema de instabilidade e superestimação do valor Q. O primeiro problema, conhecido como esquecimento catastrófico, é mostrado na Figura 2.8, em que é possível notar que o agente deixa de obter recompensas altas e precisa reapreender como maximizar a obtenção de recompensas. O segundo problema presente no DDPG, a superestimação do valor Q retornado pela rede *critic*, é adicionado ao problema de sensibilidade a hiper parâmetros iniciais.

O trabalho (FUJIMOTO; HOOF; MEGER, 2018) apresenta o algoritmo *Twin Delayed Deep Deterministic Policy Gradients* (TD3), que propõe três ajustes para resolver os problemas do DDPG mencionados anteriormente.

O primeiro ajuste é o *delayed learning*, no qual tanto a rede *actor* quanto as redes alvos são atualizadas em uma taxa mais baixa em relação à rede *critic*. Isso permite que a rede *critic* tenha mais tempo para se ajustar antes que sua saída seja utilizada como perda (*loss*) para a rede *actor*.

Além disso, como a rede *actor target* também segue uma política determinística para prever a próxima ação, é adicionado um ruído ξ à próxima ação a_{t+1}. Isso aumenta a robustez aos hiper parâmetros iniciais e é chamado de *target exploration*. Essa abordagem é ilustrada na Equação 2.38, com $a_{t+1} = \mu_{targ}(s_{t+1}) + \xi$, utilizada no treinamento do agente.

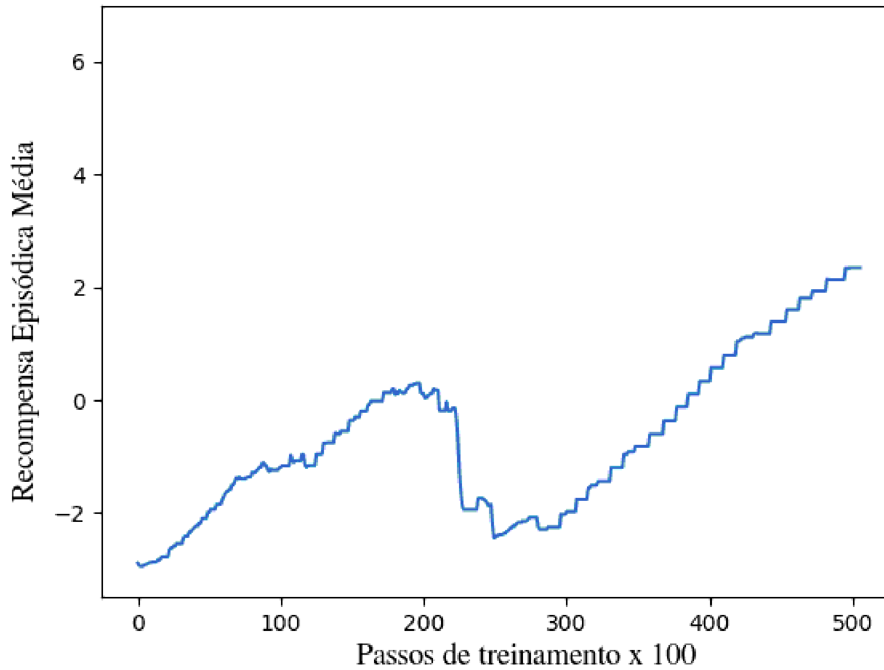

Figura 2.8 – Esquecimento catastrófico em agentes de aprendizado por reforço.
Fonte: Andrea Asperti (2008)

$$h_t = r_t + \gamma min(\hat{Q}_{targ1}(s_{t+1}, \mu_{targ}(s_{t+1}) + \xi), \hat{Q}_{targ2}(s_{t+1}, \mu_{targ}(s_{t+1}) + \xi)). \tag{2.38}$$

Por fim, o algoritmo TD3 utiliza duas redes *critics* e duas redes *target critics* para mitigar a superestimação do valor Q. A abordagem utilizada é chamada de *clipped double learning*, na qual o menor valor Q entre as duas redes *critics* é selecionado. Isso é feito para reduzir a tendência de superestimar os valores Q. Como resultado, a Equação 2.37 é reescrita como

$$L_{critic1} = \frac{1}{N}\sum_{t=0}^{T}(h_t - \hat{Q}_1(s_t, a_t))^2 \tag{2.39}$$

e

$$L_{critic2} = \frac{1}{N}\sum_{t=0}^{T}(h_t - \hat{Q}_2(s_t, a_t))^2 \tag{2.40}$$

A política da rede *actor* é atualizada seguindo a Equação 2.28, mas alterando o índice do valor Q para Q_1, visto que agora são duas redes *critics*. Apenas a primeira rede *critic* é usada para a geração do gradiente, conforme

$$\nabla_{w^\mu} J(w) = E_\mu[\nabla_\mu Q_1(s_t, \mu(s_t))\nabla_{w^\mu}\mu(s_t)]. \tag{2.41}$$

2.4 Futebol de robôs

O futebol de robôs é uma categoria da competição internacional LARC (*Latin American Robotics Competition*) que ocorre anualmente. Nessa categoria, chamada de *Very Small Size Soccer* (VSSS)[1], dois times de robôs competem, sendo que cada time é composto por 3 robôs. Esses robôs devem ter dimensões que os permitam caber em um cubo de 75×75 mm, como ilustrado na Figura 2.9. Além disso, eles devem ser capazes de jogar futebol de forma autônoma (BASSANI et al., 2020).

Figura 2.9 – Categoria VSSS de futebol de robôs.
Fonte: Pena Mateus Machado (2020)

Cada jogo consiste em duas partidas, sendo que cada partida tem a duração de 5 minutos. O time que marcar mais gols vence (BASSANI et al., 2020). As regras relacionadas a faltas, prorrogações e pênaltis para desempates não são abordadas no contexto do presente trabalho.

A dinâmica do jogo é ilustrada na Figura 2.10, onde a câmera captura o estado atual do jogo no ambiente. Em seguida, esse quadro é enviado para o algoritmo de visão, que extrai informações sobre as posições e velocidades de todos os robôs e da bola no campo. Com base nessas informações, um algoritmo de estratégia é utilizado para determinar as posições e ângulos que cada robô do time deve assumir, conhecidos como alvos (KIM et al., 2004).

Os alvos desejados são passados para o controle, que é responsável por definir as velocidades angulares e lineares necessárias para que os robôs atinjam as posições desejadas

[1]As regras completas estão disponíveis em: http://200.145.27.208/cbr/wp-content/uploads/2020/07/vssRules__English_.pdf

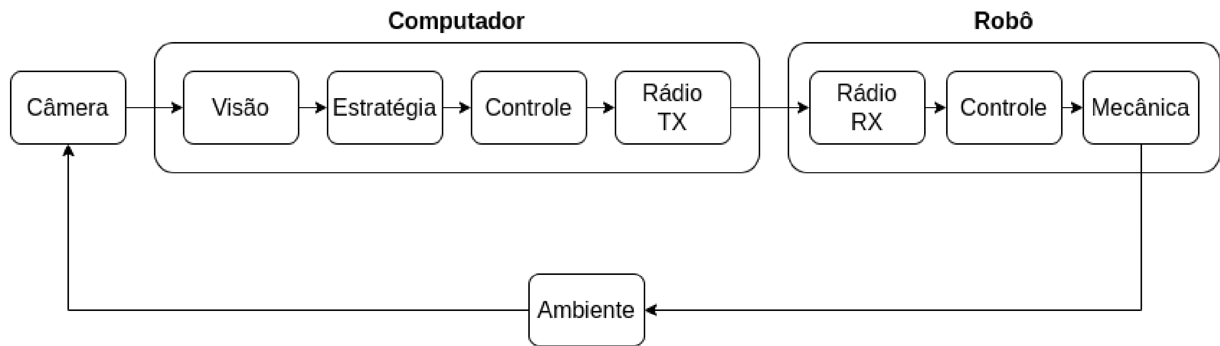

Figura 2.10 – Diagrama das partes presentes no futebol de robôs.
Fonte: Próprio autor

(x,y,θ). Os valores x denotam a posição do robô ao seguir um sistema cartesiano com centro no meio do campo, onde x é paralelo a horizontal e y paralelo a vertical. O valor θ denota o ângulo do robô em relação à horizontal, sendo θ positivo no sentido anti-horário.

Esse processo é realizado no computador e as velocidades desejadas são transmitidas por meio de rádio para um receptor no robô. Uma vez obtidas as velocidades desejadas, cada robô utiliza um controle interno para determinar os níveis de tensão que devem ser aplicados aos motores, a fim de alcançar as velocidades desejadas.

Os objetos presentes no campo, como robôs, bola e gols, são representados por vetores, conforme ilustrado na Figura 2.11. A posição da bola é denotada por $P_b = (x,y)$, a posição do gol inimigo é denotada por $P_g = (x,y)$, e a posição de um robô qualquer é denotada por $P_r = (x,y,\theta)$, onde θ é o ângulo no sentido anti-horário.

O vetor velocidade linear do robô, denotado por $\vec{v}$, é calculado por

$$\vec{v} = v \cdot (\hat{\imath} \cdot \cos\theta + \hat{\jmath} \cdot \sin\theta). \tag{2.42}$$

O robô só pode realizar movimentos de translação na sua componente y}. Isso ocorre devido à cinemática do robô, que não permite movimentos de translação na direção x', mostrado na Figura 2.11. A velocidade angular do robô é representada por $w = \dot{\theta}$ (KIM et al., 2004).

A cada robô pode ser atribuída uma entidade específica, sendo elas atacante, goleiro e zagueiro, totalizando 3 entidades disponíveis. É possível ter até 2 zagueiros ou 2 atacantes, mas não é permitido ter dois goleiros simultaneamente (KIM et al., 2004).

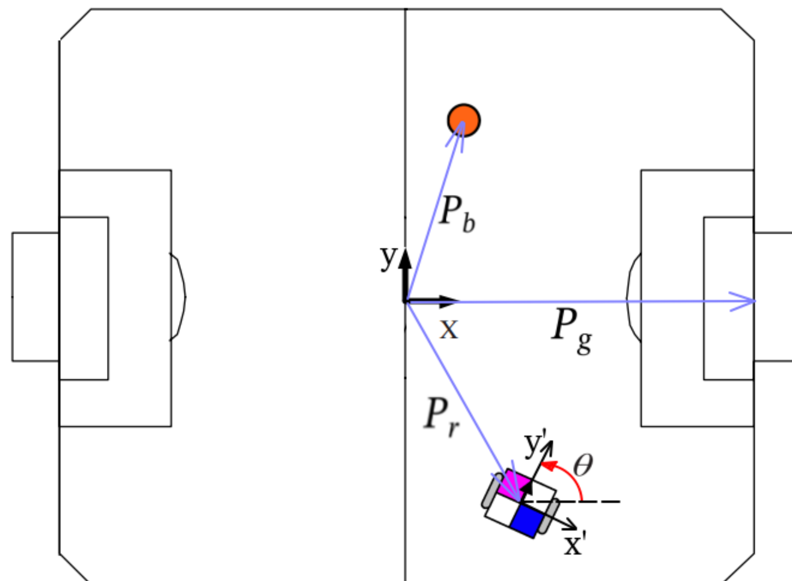

Figura 2.11 – P_g denota a posição do gol adversário, P_r denota a posição do robô e P_b a posição da bola. A informação do ângulo do robô é presente em P_r.

Adaptado de: Kim et al. (2004, p. 152)

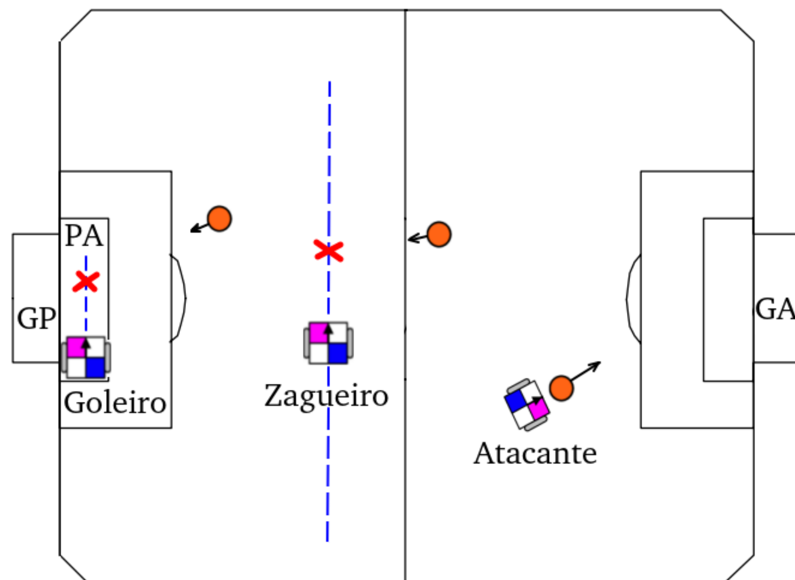

Figura 2.12 – Atribuições das entidades no futebol de robôs. O goleiro atua na pequena área (PA) para evitar que a bola alcance o gol de sua própria equipe (GP). O zagueiro atua em seu próprio campo, impedindo a progressão da bola em direção ao gol aliado. O atacante tenta levar a bola em direção ao gol adversário (GA).

Adaptado de: Kim et al. (2004, p. 152) e Pena Mateus Machado (2020)

A função do goleiro é proteger a pequena área (PA) mostrada na Figura 2.12 e evitar que a bola entre no gol de sua equipe (GP). O zagueiro, por sua vez, atua no lado do time aliado, impedindo que a bola avance em direção ao gol aliado. O atacante pode atuar tanto

na área do zagueiro quanto na área do goleiro, mas geralmente sua principal função é levar a bola em direção ao gol adversário (GA) mostrado na Figura 2.12. Para simplificar o algoritmo de estratégia, é comum ter 1 goleiro e 2 atacantes (KIM et al., 2004).

Na Figura 2.10, o bloco de estratégia pode ser subdividido em três algoritmos, cada um responsável por uma entidade específica. Esses algoritmos, conhecidos como controladores de entidade, têm como objetivo definir o alvo (x, y, θ) para o robô associado a eles (KIM et al., 2004).

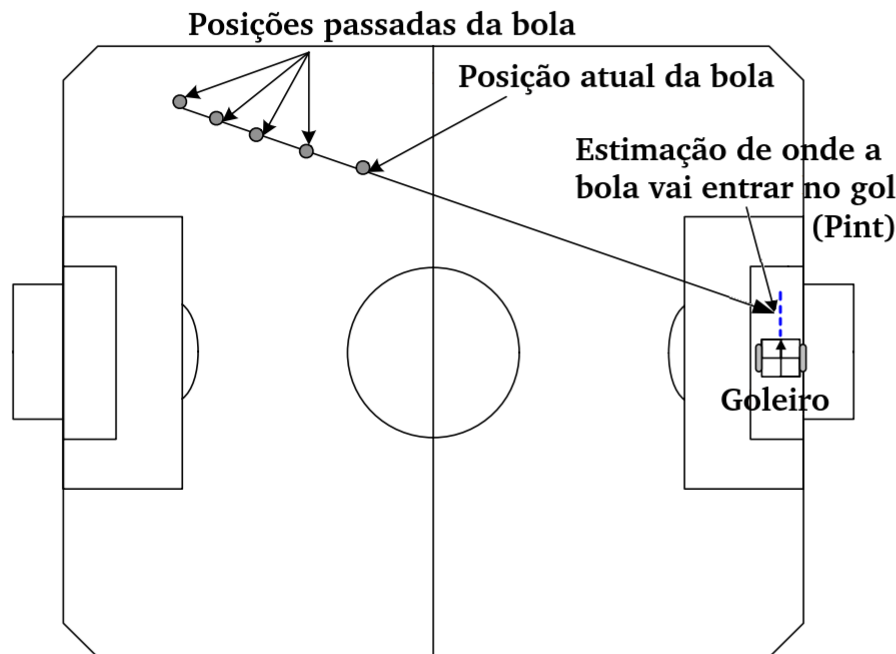

Figura 2.13 – Estimação de onde o goleiro deve interceptar a bola.
Adaptado de: Kim et al. (2004, p. 157)

Uma abordagem clássica para o controlador da entidade goleiro é descrita em Kim et al. (2004, p. 115). Nessa abordagem, o objetivo é estimar o ponto de interceptação da bola quando ela está prestes a entrar no gol, representado como P_{int} e ilustrado na Figura 2.13. Para isso, utiliza-se a informação de que o goleiro está sempre em um ângulo fixo de 90° e move-se apenas para frente e para trás para bloquear as bolas. Além disso, considera-se que a bola possui um vetor de velocidade V_b e um vetor de posição P_b. Dessa forma, a posição y em que o goleiro deve interceptar a bola é ilustrada na Figura 2.14 e dada por

$$P_b + t \cdot V_b = P_{int}, \tag{2.43}$$

onde $P_{int} = (x, y)$ com x conhecido.

Ao separar as componentes dos vetores da Equação 2.43, resulta no sistema de equações lineares definido por

$$\begin{cases} P_{bx} + t \cdot V_{bx} & = P_{intx}, \\ P_{by} + t \cdot V_{by} & = P_{inty}. \end{cases} \tag{2.44}$$

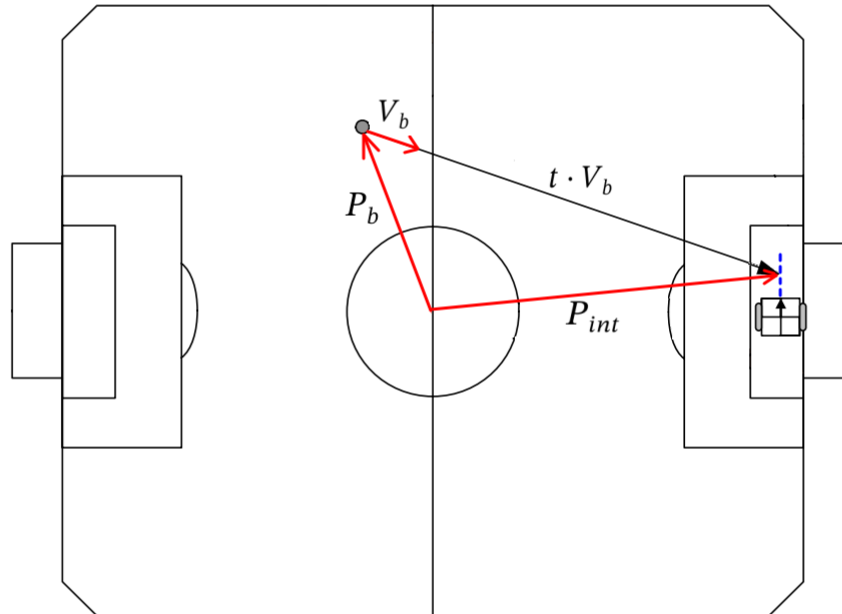

Figura 2.14 – Modelagem matemática da estimativa de onde a bola vai entrar no gol.
Adaptado de: Kim et al. (2004, p. 157)

Uma vez que o valor de P_{intx} da Equação 2.44 já é conhecido, então P_{inty}, que é a componente y do ponto de interceptação da bola com o goleiro (KIM et al., 2004), é dado por

$$P_{inty} = P_{by} + V_{by} \cdot \frac{P_{intx} - P_{bx}}{V_{bx}}. \tag{2.45}$$

A partir da Equação 2.45, obtêm-se uma entidade goleiro que consistem em movê-lo sempre para o ponto $P = (x, P_{inty})$, onde o valor de x é determinado pela posição x fixa do goleiro. As abordagens comumente utilizadas para o algoritmo controlador da entidade atacante são discutidas na seção 2.6.

2.5 Simulador

Com o propósito de acelerar a obtenção dos parâmetros da política ótima $\pi(s_t)$, os blocos de câmera, visão, rádio tx, rádio rx, controle, mecânica e ambiente mencionados na Figura 2.10, são substituídos por um simulador que reproduz a dinâmica do ambiente de futebol de robôs.

Entre os simuladores candidatos destacam-se: o oficial da da categoria VSSS, denominado FIRACup *Soccer Simulator* (FIRASim) (BASSANI et al., 2020), o rSoccer (MARTINS et al., 2021) e o PythonSimulator (KALEJAIYE, 2019).

O FIRASim [2] é baseado no simulador grSim que utiliza a biblioteca *Open Dynamics Engine* (ODE) para a simular as dinâmicas e a biblioteca Protobuf3 do Google para a comunicação com o ambiente, como mostrado na Figura 2.15.

Figura 2.15 – Simulador FIRASim.
Fonte: Pena Mateus Machado (2020)

A comunicação com o ambiente é realizada por meio de sockets, onde as velocidades desejadas de cada roda dos robôs (v_l e v_r) são enviadas para o simulador. O ambiente envia um quadro (*frame*) F_t como saída para o endereço IP e porta pré-configurados. Um quadro F_t consiste em informações de posição e velocidade de todos os robôs e da bola dentro do campo. No entanto, o FIRASim apresenta a desvantagem de não oferecer um *framework* que siga os padrões dos ambientes do *OpenAI Gym*, ou seja, não fornece uma função de recompensa baseada em um estado e conjunto de ações realizadas.

O RSoccer é um simulador desenvolvido pela equipe de competição RoboCin em 2021, utilizando a linguagem Python. Ele é baseado no FIRASim, mas apresenta um *framework* que segue os padrões do OpenAI *Gym*. O RSoccer propõe um ambiente baseado em um processo de decisão de Markov como simulador para o treinamento de redes neurais, visando

[2] Código disponível em: `https://github.com/robocin/FIRASim`

uma maior proximidade com o ambiente real em comparação ao FIRASim. Esse ambiente é interessante quando se deseja transferir o agente de um ambiente simulado para o controle de um robô físico. O trabalho (MARTINS et al., 2021), o qual utiliza o simulador RSoccer para o treinamento do agente, propõe a utilização de uma ANN do tipo *multilayer perceptron* (MLP) para relacionar as velocidades simuladas e as velocidades reais, realizando a transferência de domínios (BASSANI et al., 2020).

O PythonSimulator[3] é um simulador desenvolvido pela equipe UnBall em 2020 utilizando as bibliotecas PyGame e PyBox2D para a simulação das dinâmicas do ambiente. Ele oferece a opção de interação através de um processo de decisão de Markov e tendo sido usado em (KALEJAIYE, 2019).

2.6 Trabalhos relacionados

Como discutido, existe uma proposta eficiente para o algoritmo da entidade goleiro capaz de defender o gol, porém ainda são apresentadas várias propostas para desenvolver um algoritmo eficiente para o controlador para a entidade atacante.

O controlador da entidade atacante se baseia em dois princípios: primeiro, deslocar-se até a posição atrás da bola em um ângulo alinhado com o gol adversário, conforme ilustrado na Figura 2.16; em seguida, empurrar a bola em direção ao gol adversário, conhecido como chute, como mostrado na Figura 2.17.

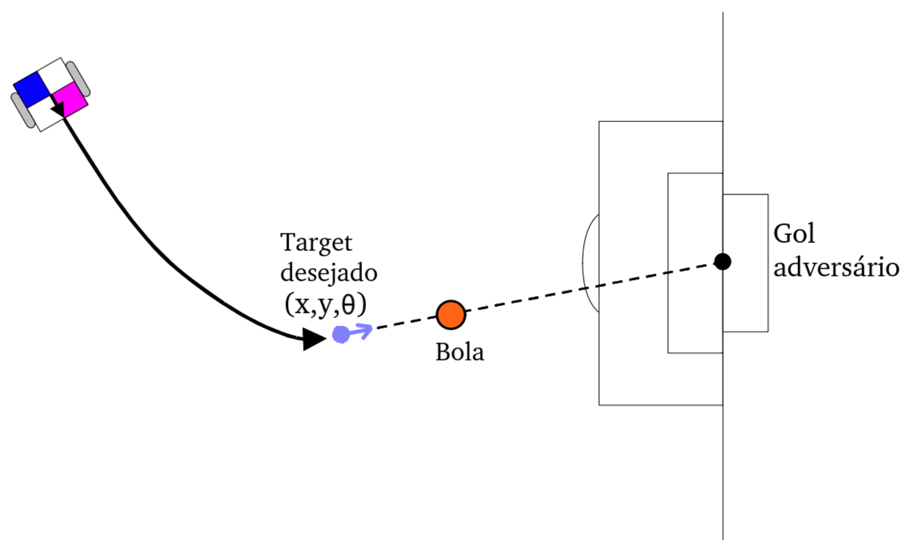

Figura 2.16 – Modelagem matemática da estimativa de onde a bola vai entrar no gol.
Adaptado de: Kim et al. (2004, p. 113)

[3] Código disponível em: `https://github.com/unball/python_simulator`

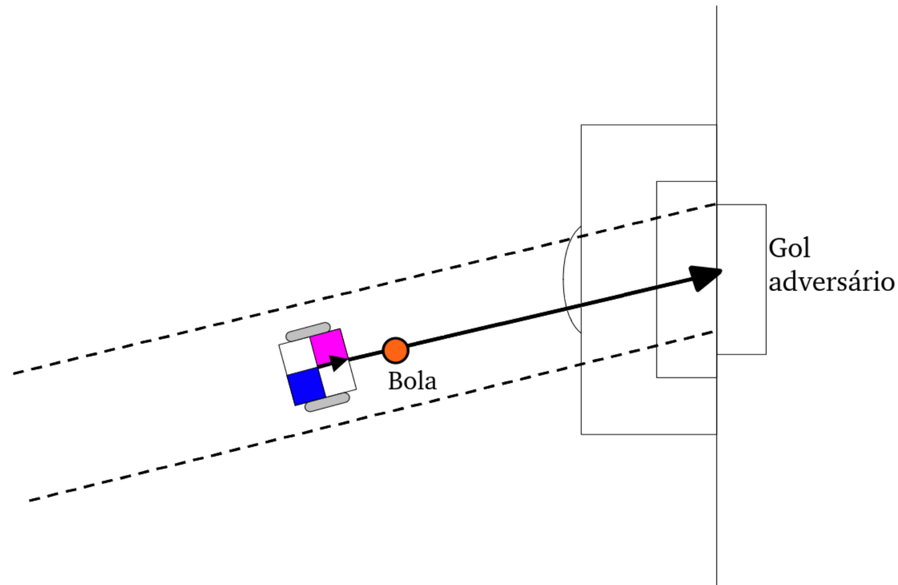

Figura 2.17 – alvo desejado para o robô que possui a entidade atacante.
Adaptado de: Kim et al. (2004, p. 113)

Para simplificar a implementação de alguns algoritmos, os vetores dos objetos dentro do campo podem ser decompostos em vetores relativos e seus respectivos ângulos. Os vetores relativos são mostrados na Figura 2.18.

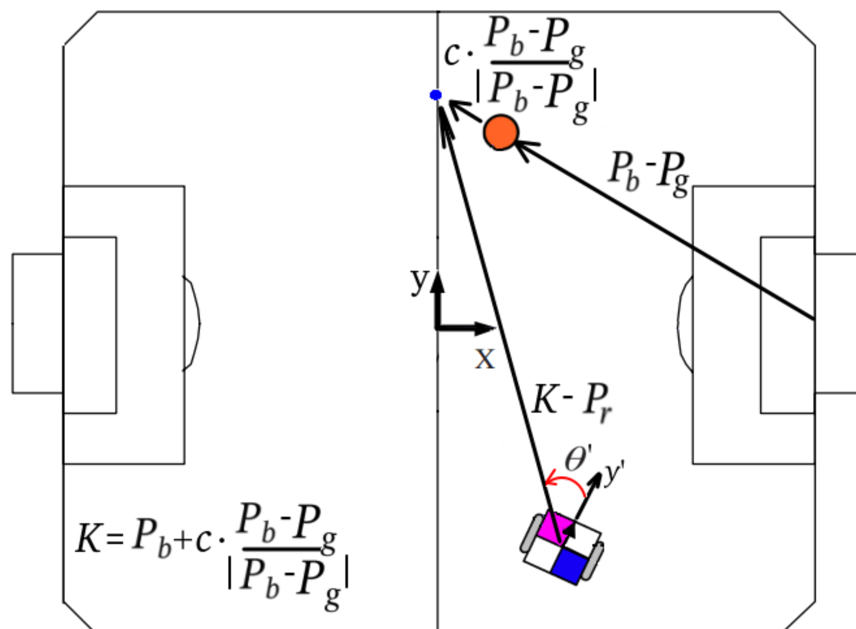

Figura 2.18 – O vetor unitário de $P_b - P_g$ multiplicado com uma constante c pode ser utilizado para estimar a posição desejada K atrás da bola na qual o robô fique alinhado com o gol adversário.
Adaptado de: Kim et al. (2004, p. 152)

Para que o robô atacante possa realizar o movimento de chute, é necessário determinar um alvo, ou seja, a posição (x, y, θ) atrás da bola que o alinhe com o gol adversário. Uma estimativa inicial para esse problema é ilustrada na Figura 2.18, onde o vetor relativo $P_b - P_g$ é utilizado, juntamente com um deslocamento adicional. No entanto, essa abordagem apresenta a desvantagem de introduzir um ruído na posição k quando a bola está em movimento. Isso resulta em um problema de interceptação de um alvo em movimento, em que tanto o interceptador quanto o objeto estão se deslocando.

O trabalho (MAKAROV et al., 2019) aborda o problema de interceptar a bola em movimento na categoria de futebol de robôs *Small Size Soccer* (SSS). Nessa categoria, os robôs podem realizar movimentos de translação ortogonal na direção x'.

No contexto da categoria *Very Small Size* (VSSS), o trabalho (MEDEIROS; MARCOS; YONEYAMA, 2020) propõe uma abordagem para a interceptação da bola em movimento utilizando a entidade zagueiro, quando a bola está se aproximando do gol aliado. Nesse trabalho, é empregado o aprendizado por reforço em conjunto com a técnica de *curriculum learning* (SOVIANY et al., 2022). O objetivo do *curriculum learning* é permitir que o agente comece com ações mais simples e, gradualmente, aumente a complexidade. No caso específico do robô zagueiro, o artigo (MEDEIROS; MARCOS; YONEYAMA, 2020) propõe treinar o robô para interceptar a bola começando com uma posição inicial próxima a ele e, em seguida, aumentando a distância e velocidade da bola ao longo do treinamento.

Uma vez definido o alvo, surge o desafio de determinar como o robô irá alcançá-lo, considerando que existem infinitas trajetórias possíveis entre a posição atual do robô e a posição desejada (x, y, θ). Para resolver esse problema, o trabalho (TU, 2002) propõe o uso do algoritmo Dubins para o planejamento de rotas.

O Dubins é um algoritmo de planejamento de menor rota que conecta dois pontos A e B, levando em consideração restrições nos ângulos de saída e chegada e nas curvaturas das trajetórias, como ilustrado na Figura 2.19. Embora seja amplamente utilizado no planejamento de rotas em áreas como robótica, aviação e automobilística, o Dubins possui uma limitação. A menor rota entre A e B pode variar significativamente em intervalos de tempo próximos, especialmente se o ponto B também estiver em movimento. A Figura 2.20 mostra como o ângulo de chegada b varia em função da posição da bola e do campo adversário, sendo V_b a velocidade da bola.

O trabalho (LIM et al., 2008) apresenta uma alternativa ao método proposto em (TU, 2002) por meio da aplicação de um campo vetorial denominado UVF (*UniVector Field*) para objetos estáticos e em movimento. O UVF é um campo estático que possui a vantagem de não exigir a posição do robô para ser gerado.

O campo vetorial UVF é gerado a partir de duas espirais hiperbólicas, como ilustrado na Figura 2.21. O ângulo desejado para o robô é representado pelo ângulo destacado em

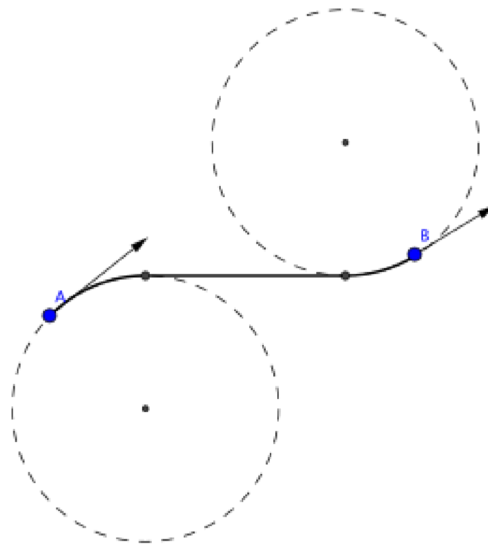

Figura 2.19 – Dubins.
Adaptado de: Kim et al. (2004, p. 152)

vermelho no campo, permitindo que o robô se alinhe com a bola. Nesse caso, o controle é responsável por encontrar as velocidades V_r (linear) e W (angular) que minimizam o erro de ângulo entre o robô e o campo.

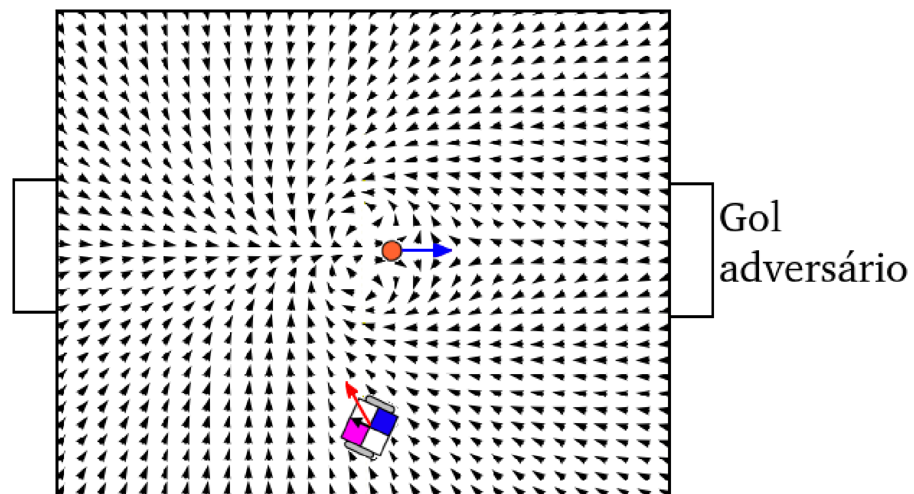

Figura 2.21 – O vetor unitário em (x,y), denotado pela cor vermelha, representa o ângulo do campo desejado para o robô na posição x,y e de forma que caso alinhe o ângulo dele com os dos campo o torna possível chegar na bola alinhado com o veto em azul.
Adaptado de: Kim et al. (2004)

O trabalho (KALEJAIYE, 2019) aborda o treinamento de um robô atacante usando aprendizado por reforço, sem a necessidade de definir um caminho específico até um alvo ou um alvo em si. Para isso, é proposta a integração do bloco de controle à estratégia, permitindo

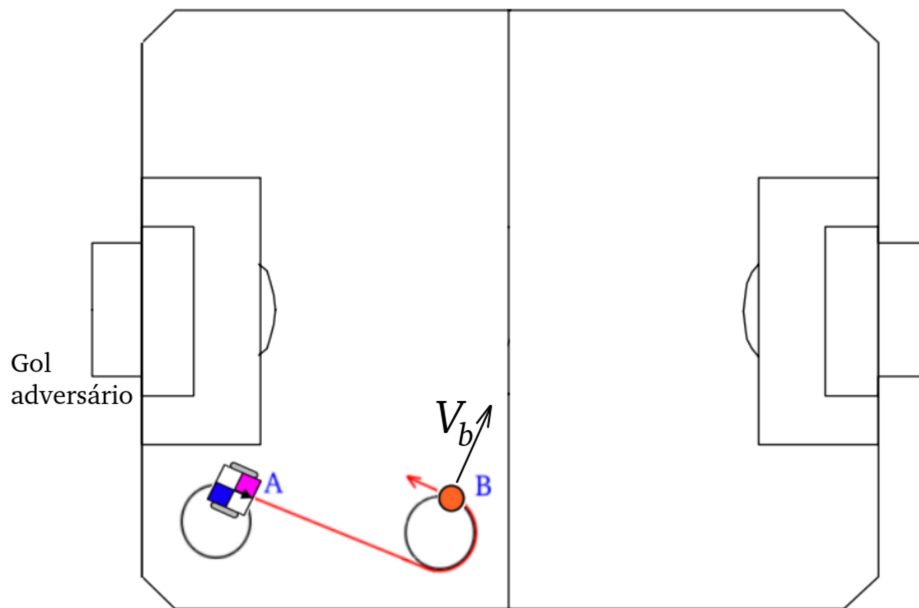

Figura 2.20 – O ponto A é representado pelo robô e o ponto B pela bola com o ângulo de chegada dado em função do vetor relativo de posição da bola até o gol. V_b é a velocidade da bola. Adaptado de: Kim et al. (2004)

que a rede neural de controle forneça a velocidade angular e linear desejada como saída.

É importante destacar que (KALEJAIYE, 2019) também apresenta sua própria função de recompensa. No entanto, uma desvantagem principal do trabalho é a necessidade de discretização das velocidades de saída V_r (linear) e W (angular) da rede. Isso significa que a ação tomada pelo agente é selecionada entre um conjunto de opções discretas, usando a função *softmax*.

A estratégia da UnBall[4] é baseada em uma máquina de estados, na qual um campo UVF (*Univector Field*) (LIM et al., 2008) é selecionado de acordo com a configuração do campo. Quando o robô possui o mesmo ângulo que o campo em uma determinada posição, ele é direcionado para alcançar o alvo desejado, que, nesse caso, é a bola (conforme ilustrado na Figura 2.21). Ao se aproximar da bola, o robô é redirecionado para um novo campo UVF, com o objetivo de empurrar a bola em direção ao gol adversário. A estratégia da UnBall utiliza uma lei de controle que recebe o ângulo desejado para o robô na respectiva posição, determinando a velocidade linear e angular necessárias para seguir a referência.

[4]Podem ser obtidas mais informações acerca da equipe via: http://ft.unb.br/index.php?option=com_content&view=article&id=13&Itemid=123

2.7 Conclusão do capítulo

Neste capítulo, fora apresentados os conceitos teóricos relacionados ao aprendizado por reforço, um campo que tem ganhado destaque nos últimos anos, uma vez que tem alcançado ou até mesmo superado pontuações humanas em vários jogos, substituindo o antigo método tabular conhecido como *Q-table* por redes neurais. O algoritmo DDPG tem sido amplamente utilizado em diversas aplicações, incluindo o futebol de robôs, que é uma área com poucos trabalhos publicados, apesar de apresentar vários desafios a serem enfrentados.

Como discutido, o goleiro já possui um algoritmo simples, porém eficaz, capaz de cumprir sua função de evitar que a bola entre no gol. Esse algoritmo se baseia na estimativa futura da posição em que o robô e a bola irão se encontrar, e direciona o goleiro para essa posição. No entanto, ainda há muito espaço para o desenvolvimento de trabalhos relacionados à estratégia que gerencia o atacante, uma vez que é uma parte do jogo que requer atenção e melhorias.

3 Proposta

Para maximizar a quantidade de gols feitos no futebol de robôs, várias propostas foram desenvolvidas. No caso das propostas que envolvem máquinas de estado, é necessário implementar um atacante capaz de tomar decisões com base nas diferentes situações durante o jogo. Essas situações precisam ser identificadas pelos desenvolvedores e traduzidas em ações para o robô. No entanto, essa abordagem tem algumas desvantagens, como a necessidade de ajustar vários parâmetros e lidar com incertezas, como erros de interceptação da bola em momentos futuros.

Este trabalho apresenta uma nova estratégia que visa a substituição da arquitetura complexa atualmente utilizada, a qual depende de múltiplos algoritmos e controle para viabilizar o movimento do atacante. A proposta de melhoria é fundamentada no estudo realizado por (KALEJAIYE, 2019), porém, diferentemente daquele estudo, busca-se substituir as ações quantizadas de velocidade linear e angular (V e W) por ações contínuas. Alinhado com os objetivos de (KALEJAIYE, 2019), este trabalho também busca capacitar um único agente a interagir com a bola dentro do campo. Tais modificações têm como propósito simplificar o controle do atacante e proporcionar maior flexibilidade em suas ações.

3.1 Informações de entrada dos agentes

Para permitir a interação do agente no ambiente, propomos a utilização dos algoritmos DDPG (*Deep Deterministic Policy Gradient*) e TD3 (*Twin Delayed DDPG*). A fim de viabilizar o treinamento, este trabalho avalia o desempenho do agente ao receber o estado S_{rel} em comparação com o estado S_{abs}. O estado S_{rel} é composto principalmente por vetores relativos e segue a seguinte ordem:

- Ângulo θ' entre a velocidade do robô V_r e o vetor de distância relativa do robô até a bola $(P_b - P_r)$
- Norma do vetor de distância relativa do robô até a bola $(|P_b - P_r|)$
- Vetor velocidade linear do robô V_r
- Velocidade angular W do robô dividida pela máxima velocidade angular possível (6.28 rads/seg)
- Angulo θ do robô
- Distância do robô até cada uma das quatro paredes do campo, a fim de evitar colisões

- Distância do robô até o gol adversário ($|P_g - P_r|$)
- Ângulo entre a velocidade linear do robô V_r e o vetor distância entre o robô e o gol adversário ($|P_g - P_r|$)

Por sua vez, o estado S_{abs} é composto principalmente por vetores absolutos e segue a seguinte ordem:

- Posição (x, y, θ) do robô
- Posição (x, y) da bola
- Posição (x, y) do gol
- Velocidade linear V_r do robô
- Velocidade angular W do robô dividida pela máxima velocidade angular possível (6.28 rads/seg)

3.2 Interação com o ambiente

Como este trabalho tem como objetivo inicial uma aplicação na categoria simulada da competição, utilizamos o simulador oficial da categoria, o FIRASim. Treinando o agente nesse ambiente, não será necessário um trabalho subsequente para a transição para outros domínios.

Dado que o simulador FIRASim não fornece diretamente a recompensa obtida pelo agente ao final de cada passo t, foi necessário criar uma classe que permitisse a comunicação do agente com o ambiente, seguindo uma arquitetura da OpenAI para o treinamento de agentes por reforço. Esse algoritmo recebe o estado S_{t+1} do simulador por meio de sockets UDP, calcula a recompensa obtida pelo agente ao realizar a ação a_t no estado s_t e fornece R_t, S_t e S_{t+1}. Onde S_{t+1} representa o estado mais recente retornado pelo simulador, que é resultado da ação a_t realizada pelo agente.

Cada época de treinamento é definida como um intervalo de 20 segundos. Após esse período, um novo estado inicial é gerado com a posição da bola e do robô definidos aleatoriamente. Essa duração foi escolhida para limitar a quantidade de ações possíveis do agente durante esse intervalo de tempo, simplificando assim o trabalho de previsão do *Q-value* estimado pela rede *critic*. Ao reiniciar o estado após os 20 segundos, esse tempo resolve situações em que o robô fica preso nas paredes. Além disso, evita que o robô fique apenas movendo a bola pelo campo para maximizar a recompensa sem realmente tentar marcar um gol. Uma janela de tempo muito curta poderia não dar tempo suficiente para o robô concluir a ação de marcar um gol, prejudicando assim a aprendizagem da importância

dessa ação. A sincronização entre o agente e o ambiente é realizada por meio da classe, que coleta informações do simulador a cada intervalo de 30 milissegundos.

3.3 Função de recompensa

O presente trabalho utiliza a função de recompensa R_t proposta por (MARTINS et al., 2021) para ambientes de ambientes de futebol de robôs, definida como

$$R(t) = w_m R_m + w_p R_p + w_e R_e + w_g R_g \tag{3.1}$$

$$R(t) = 0.2 \cdot R_m + 0.8 \cdot R_p + 2 \cdot 10^{-3} \cdot R_e + R_g \tag{3.2}$$

onde w_m, w_p, w_e, w_g, R_m, R_R, R_e e R_g são:

- w_m, w_p, w_e, w_g são constantes com valores respectivos de 0.2, 0.8, 2×10^{-3} e 1. Sendo os valores obtidos de (MARTINS et al., 2021).
- R_m é a recompensa parcial obtida pelo agente ao se aproximar ou se afastar da bola. É calculada projetando o vetor velocidade do robô V_r no vetor unitário da posição relativa do robô até a bola $(P_b - P_r)$, conforme mostrado na Figura 3.22
- R_p é a recompensa parcial obtida caso a bola se afaste ou se aproxime do gol adversário. É calculada projetando o vetor velocidade da bola V_b no vetor unitário da posição relativa da bola até o gol $P_g - P_b$, conforme ilustrado na Figura 3.23
- R_e é uma penalidade proporcional à energia linear e angular utilizada pelo agente, representada por $V_r^2 + W_r^2$. Essa recompensa negativa é proposta por (MARTINS et al., 2021) para desencorajar movimentos desnecessários do robô
- R_g é a recompensa com valor 1 caso a bola entre no gol adversário e -1 caso a bola entre no próprio gol

3.4 *Experience Replay*

Em algoritmos de aprendizado por reforço como DDPG, A2C, TD3 e outros, é necessário treinar a rede do agente utilizando estados passados. Para isso, o trabalho (ZHANG, S.; SUTTON, 2017) introduz um bloco de memória chamado *Experience Replay* ou *Replay Buffer*, que armazena diferentes estados do ambiente. Cada entrada nesse bloco é representada por uma tupla ordenada $e_t =< s_t, a_t, r_t, s_{t+1} >$, contendo o estado s_t, a ação a_t, a recompensa

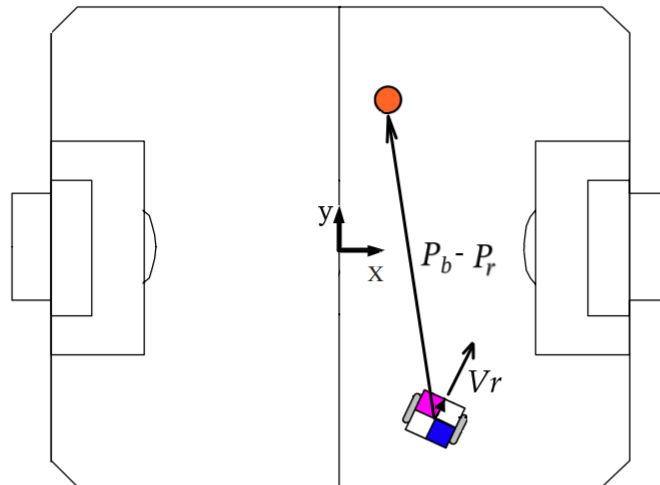

Figura 3.22 – A projeção do vetor de velocidade do robô V_r sobre o vetor unitário que aponta da posição do robô P_r até a posição da bola P_b indica se o robô está se aproximando ou se afastando da bola. Essa projeção é utilizada na função de recompensa R_t para determinar a pontuação do agente.

Adaptado de: Kim et al. (2004)

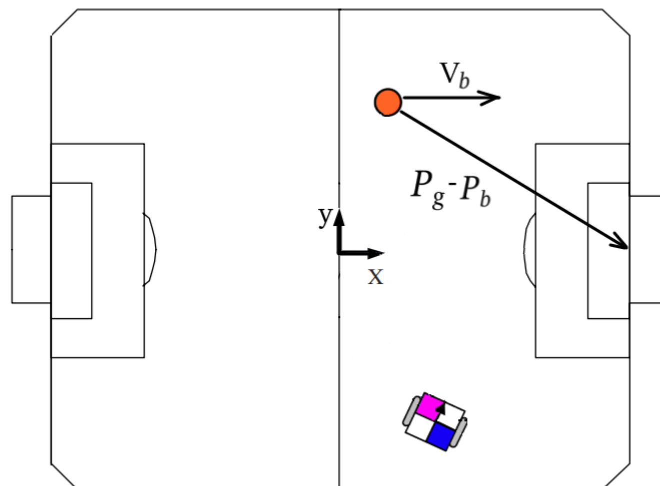

Figura 3.23 – A projeção do vetor de velocidade da bola V_b sobre o vetor unitário que aponta da posição da bola P_b até a posição do gol P_g indica se a bola está se aproximando ou se afastando do gol. Essa projeção é utilizada na função de recompensa R_t para determinar a pontuação.

Adaptado de: Kim et al. (2004)

r_t e o próximo estado s_{t+1} no instante t. Esse mecanismo permite ao agente aprender com experiências passadas e melhorar seu desempenho ao longo do tempo.

Uma vez que os estados coletados pelo agente durante a interação com o ambiente não são independentes entre si, é necessário amostrar o bloco de memória e para obter diferentes experiências e_ζ, onde $\zeta \in \mathbb{Z}$, que sejam distintas umas das outras. Isso permite o treinamento efetivo do agente.

Neste trabalho, foi utilizado um bloco de memória e com tamanho de 6.666.666 passos de tempo, o que é suficiente para armazenar as últimas 10 mil épocas em que o agente interagiu com o ambiente. Essa escolha de uma memória ampla se deve ao problema de esquecimento catastrófico, que pode ocorrer durante o treinamento, embora seja amenizado no TD3. Para o treinamento das redes neurais, foi adotado um tamanho de lote (*batch size*) de 256 amostras.

Uma variação do *Replay Buffer*, conhecida como *Prioritized Experience Replay* (PER), prioriza a amostragem de estados com recompensas mais altas com maior frequência para a rede neural. A ideia por trás dessa abordagem é permitir que a rede aprenda mais rapidamente a repetir essas recompensas mais altas, como demonstrado em (HOU; ZHANG, Y., 2019). No entanto, devido à necessidade de ajustes de hiperparâmetros para a utilização efetiva do PER, optou-se por continuar utilizando uma amostragem uniforme no conjunto de dados (*dataset*).

3.5 Arquitetura das redes neurais e hiperparâmetros

Durante a interação do agente com o ambiente, é necessário o uso de uma rede neural. No entanto, devido à falta de uma arquitetura estabelecida para a aplicação de aprendizado por reforço no contexto de futebol de robôs, o foco deste trabalho foi buscar artigos que oferecessem diretrizes sobre como estruturar a rede neural para maximizar o aprendizado do agente.

A camada de *batch normalization*, proposta por (IOFFE; SZEGEDY, 2015), é conhecida por sua eficácia em reduzir o tempo de treinamento de redes neurais supervisionadas, estabilizando os gradientes. Esse método se propõe a normalizar os dados de entrada da rede neural assumindo uma distribuição normal, o qual subtrai pela média do *bacth* e divide pelo desvio padrão do *batch*. No entanto, ao utilizar informações de média e desvio padrão dos dados de entrada, pressupõe-se que todas as entradas da rede possuem a mesma distribuição. Considerando que tanto o algoritmo DDPG quanto o TD3 utilizam estados e ações provenientes de diferentes políticas π armazenados em seus *replay buffers*, as distribuições dos dados são distintas. Portanto, a utilização de uma camada de *batch normalization* tende a prejudicar o desempenho do agente, conforme mencionado em (BHATT et al., 2019). Por esse motivo, a camada de *batch normalization* não foi empregada na arquitetura das redes utilizadas neste trabalho.

Nos algoritmos de aprendizado por reforço, há várias decisões de ajuste de hiperparâmetros que muitas vezes não são mencionadas nos artigos de referência. Um exemplo disso é a inicialização dos pesos da rede neural. Uma inicialização adequada dos pesos do agente pode prevenir problemas como o *vanishing gradient* e o *exploding gradient*. Na ocorrência de *vanishing gradient* os gradientes da rede neural se tornam muito próximos de zero, o

que impede a rede de ajustar os seus parâmetros e assim não aprendendo. Para o caso de *exploding gradient*, os valores dos gradientes se tornam elevados de forma os parâmetros da rede começam a tender ao infinito e assim também não aprendendo. Portanto, o trabalho (ANDRYCHOWICZ et al., 2020) investiga a influência dessa inicialização nos resultados da política π gerada pelo agente. Na arquitetura do presente trabalho, todos os pesos das redes neurais foram inicializados seguindo uma distribuição normal com média 0 e desvio padrão 0.02.

Uma inicialização dos pesos próximos a zero resulta em valores de velocidade linear (V) e angular (W) também próximos a zero. Isso se assemelha à técnica de *Curriculum Learning*, proposta por (SOVIANY et al., 2022), na qual o agente é inicialmente treinado em tarefas mais simples antes de progredir para ambientes mais desafiadores. Essa abordagem é similar à forma como uma pessoa aprende a dirigir automóveis, começando com velocidades baixas e posteriormente avançando para velocidades mais altas.

A inicialização dos pesos da rede *actor* e do *critic* com valores pequenos traz vantagens adicionais. Como as ações do agente têm velocidades baixas, as recompensas também têm valores baixos, uma vez que se baseiam na velocidade de aproximação ou afastamento do robô em relação à bola. Isso implica em um valor inicial baixo para o *Q-value* e, consequentemente, a saída da rede *critic* também é baixa devido à inicialização dos pesos. Essa abordagem permite que a perda (*loss*) da rede *critic* aumente gradual e controladamente à medida que a velocidade do robô aumenta, reduzindo o risco de *exploding gradients*.

De forma auxiliar a se tentar controlar o ajuste dos pesos das redes *actor* e *critic*, utiliza-se a abordagem de *gradient clipping*, a qual limita a norma máxima que os gradientes podem ter durante o treinamento. Tal abordagem pode a um primeiro momento parecer reduzir a velocidade de aprendizagem do agente, mas o trabalho (ZHANG, J. et al., 2019) mostra como o controle do ajuste pode acelerar o treinamento ao se manter o ajuste dos pesos controlado. No presente trabalho apenas a rede $critic$ possui *gradient clipping* e com valor de 10^{-3}, ressalta-se que o valor 10^{-2} foi testado, porém a rede *critic* ainda ficava sujeita a variações abruptas no *loss* e propagando isso para a rede *actor*.

Essas estratégias de inicialização dos pesos e *gradient clipping* são adotadas para promover um treinamento mais estável e controlado, evitando problemas de *exploiding gradients* e permitindo um aprendizado mais eficiente do agente.

As funções de ativação desempenham um papel importante na arquitetura da rede neural, pois determinam a não-linearidade modelada. No caso da rede neural *actor*, as saídas desejadas para representar a velocidade linear e angular são denotadas por y_m e y_n, respectivamente, e correspondem às saídas da última camada da rede.

Para garantir que as saídas estejam no intervalo desejado, a função de ativação tangente hiperbólica pode ser uma escolha adequada. Essa função gera valores no intervalo

$[-1, 1]$, o que é desejado para representar as velocidades.

As saídas y_m e y_n são utilizadas para calcular as velocidades lineares e angulares finais do robô. A velocidade linear é obtida multiplicando y_m pela velocidade máxima linear do robô, ou seja, $v = y_m \cdot v_{max}$, onde $v_{max} = 2$ m/seg. Da mesma forma, a velocidade angular é calculada multiplicando y_n pela velocidade angular máxima do robô, $w = y_n \cdot w_{max}$, onde $w_{max} = 6.28$ rad/seg. Essas definições permitem que as saídas y_m e y_n variem positiva ou negativamente, o que possibilita ao robô mover-se para frente e para trás, assim como girar nos sentidos horário e anti-horário.

A *Rectified Linear Unit* (ReLU) (DUBEY; SINGH; CHAUDHURI, 2022) é amplamente utilizada como função de ativação em camadas escondidas, mostrada na Figura 3.24, que é definida como

$$ReLU(x) = max(0, x). \tag{3.3}$$

Para o presente trabalho, as camadas escondidas da rede neural foram configuradas com a função de ativação Mish (MISRA, 2019), mostrada na Figura 3.24, que é definida como

$$Mish(x) = x \tanh(\text{softplus}(x)), \tag{3.4}$$

em que

$$\text{softplus}(x) = \ln(1 + e^x). \tag{3.5}$$

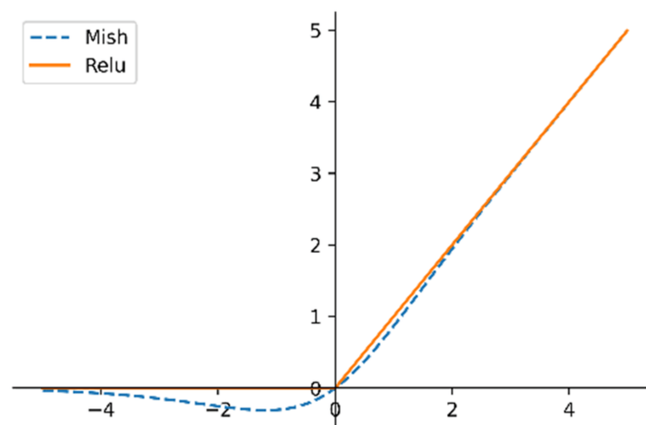

Figura 3.24 – Funções de ativação Mish e ReLU.
Fonte: Misra (2019)

A função Mish é reconhecida como uma função de ativação de última geração, demonstrando resultados superiores em relação às funções ReLU e GeLU no treinamento de redes neurais, conforme evidenciado em (MISRA, 2019). Essa escolha foi feita com base em estudos recentes que comprovaram sua eficácia.

Os otimizadores utilizados para ajustar os pesos das redes *actor* e *critic* foram configurados com os mesmos hiperparâmetros. Foi escolhido o algoritmo Adam, com uma taxa de aprendizado de 3×10^{-4}, intervalo de betas definido como $[0.9, 0.999]$, e um valor de *eps* de 10^{-8}. Esses hiperparâmetros foram adotados tanto para treinar o algoritmo DDPG quanto o TD3. Essa decisão foi tomada com o objetivo de manter uma configuração consistente e comparável entre os dois algoritmos.

Com base nas informações fornecidas, a arquitetura das redes *actor* e *critic* é definida como uma sequência de 3 camadas lineares, cada uma com 512 neurônios e a função de ativação *Mish* aplicada entre elas. Na última camada linear da rede *actor*, a função de ativação utilizada é a tangente hiperbólica, enquanto a saída da rede *critic* a função de ativação utilizada é a identidade, uma vez que o valor estimado do *Q-value* pode assumir qualquer valor, como mostrado na Figura 3.25.

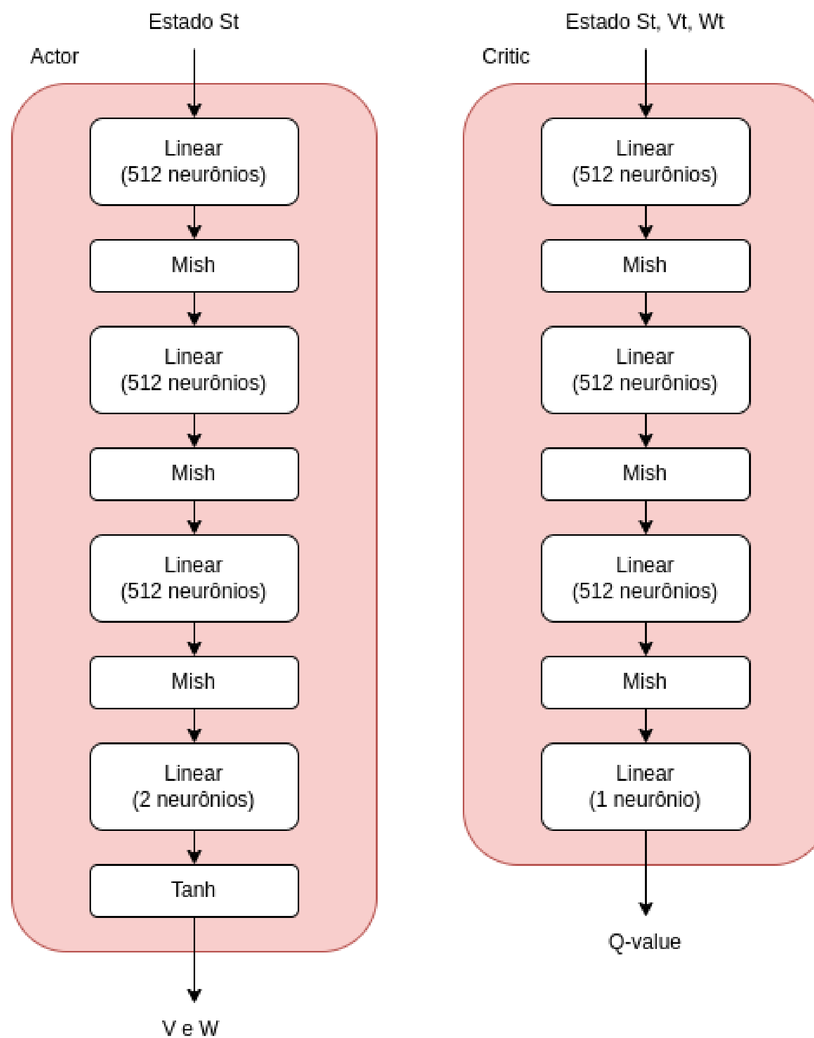

Figura 3.25 – Arquitetura da rede usada para definir o *actor* e para definir o *critic*. V e W denotam a velocidade linear a angular, respectivamente.
Fonte: Produzido pelo autor.

O parâmetro de atualização das redes *target* é definido como $\tau = 10^{-3}$. No caso do algoritmo TD3, a rede *actor* e a rede *target critic* são atualizadas a cada duas atualizações da rede *critic*.

3.6 Ruído de exploração

Devido à natureza determinística da política μ no algoritmo DDPG e TD3, é necessário adicionar um ruído ξ às ações do agente. O trabalho original do algoritmo DDPG (LILLICRAP et al., 2015) utiliza o processo estocástico de *Ornstein-Uhlenbeck* para gerar ruídos. No entanto, o artigo (FUJIMOTO; HOOF; MEGER, 2018) sugere que não há diferença significativa ao utilizar um ruído ξ gerado a partir de uma distribuição normal.

Dado esse aspecto de escolha, optou-se por gerar o ruído a partir de uma distribuição normal com média 0 e desvio padrão 0.2. Essa abordagem permite adicionar perturbações aleatórias às ações do agente durante a interação com o ambiente, auxiliando na exploração e evitando a convergência prematura para um único conjunto de ações determinísticas.

3.7 Ferramenta *ONNX Runtime*

O *ONNX Runtime* (SHRIDHAR; TOMSON; INNES, 2020) é uma ferramenta que permite a execução de modelos de redes neurais sem a necessidade de conhecer a arquitetura específica durante a fase de inferência. Essa ferramenta é especialmente útil na etapa de teste do modelo, pois permite testar diferentes modelos independentemente da arquitetura da rede utilizada no treinamento. Sem o *ONNX Runtime*, seria necessário conhecer o código da arquitetura da rede para carregar os pesos salvos. Com o uso do *ONNX Runtime*, essa dependência da arquitetura é eliminada, tornando mais flexível o teste e avaliação dos modelos treinados no presente trabalho.

3.8 Proposta de validação dos agentes

Para avaliar cada um desses modelos, foram salvos os pesos da rede a cada 10 épocas durante o treinamento. Finalizado treinamento, é escolhido o modelo com a maior recompensa obtida para a parte de validação.

Foram treinados quatro modelos distintos, cada um com uma configuração específica:

- O modelo DDPG rel utiliza o algoritmo DDPG e recebe o estado S_{rel} como entrada
- O modelo DDPG abs também utiliza o algoritmo DDPG, mas recebe o estado S_{abs} como entrada

- O modelo TD3 rel utiliza o algoritmo TD3 e recebe o estado S_{rel} como entrada
- O modelo TD3 abs utiliza o algoritmo TD3 e recebe o estado S_{abs} como entrada

Inicialmente, considerou-se uma abordagem semelhante a descrita em (MARTINS et al., 2021) para obter uma medida quantitativa de desempenho para cada um dos quatro modelos. Essa abordagem envolve a definição de mil posições aleatórias para o início do jogo e calcular o tempo necessário para cada agente marcar o primeiro gol nessas mil interações. No entanto, essa abordagem não leva em consideração a situação em que o robô ficava preso na parede e não conseguir marcar o gol. Portanto, optou-se por um método de validação diferente.

No método de validação adotado, são realizadas mil épocas de teste, onde para cada época é definida uma posição aleatória no campo de jogo. Cada época tem a duração de 1 minuto e durante esse período é medida a distância mínima que o robô conseguiu mover a bola em direção ao gol. É importante ressaltar que a bola inicia sempre no meio do campo.

Com base nesses resultados obtidos nas mil épocas, é possível estimar uma média e um desvio padrão que representam o desempenho de cada um dos modelos. Essa média e desvio padrão servem como medidas para avaliar o quão efetivo cada modelo foi em mover a bola em direção ao gol durante o teste.

O presente trabalho também realiza uma comparação com estratégias utilizadas na competição LARC. Por exemplo, no trabalho (MARTINS et al., 2021), o melhor modelo desenvolvido foi comparado com a estratégia da equipe que obteve o terceiro lugar na competição LARC 2018 em um jogo 1 contra 1. De forma a realizar uma comparação similar, a estratégia da equipe de competição UnBall, que alcançou o quinto lugar na competição LARC 2021, é utilizada como referência de comparação neste trabalho.

A estratégia da UnBall, vista na seção 2.6, é adotada como um ponto de comparação para avaliar o desempenho dos modelos desenvolvidos neste trabalho. Aspectos como eficiência em direcionar a bola ao gol adversário e o placar de jogo são considerados para a avaliação comparativa.

Portanto, neste trabalho, é realizada uma comparação numérica entre as 4 estratégias propostas e a estratégia da equipe UnBall. Essa comparação visa mensurar a eficiência em direcionar a bola ao gol adversário para cada cada estratégia. Além disso, é realizado uma partida 1 contra 1 entre a melhor rede neural treinada e a estratégia da UnBall, no qual é avaliado o saldo de gols obtido em um período de 10 minutos. Conforme as regras da LARC, cada partida tem duração de 10 minutos e o time que marcar mais gols nesse período é o vencedor.

4 Resultados

Este capítulo apresenta os resultados obtidos ao se aplicar a proposta em simulações. Os treinamentos e avaliações dos modelos foram conduzidos em uma máquina com as seguintes especificações: sistema operacional Microsoft Windows 11 Pro, processador AMD Ryzen Threadripper de 3693MHz com 32 núcleos, 256 GiB de memória RAM e placas gráficas NVIDIA GeForce RTX 3080.

Inicialmente, foi tentada uma abordagem baseada em *Curriculum Learning* semelhante à utilizada por (MEDEIROS; MARCOS; YONEYAMA, 2020). Nessa abordagem, o agente é treinado com a bola próxima para que ele entenda a relação entre a recompensa e a interação com a bola. No entanto, essa abordagem resultou em um comportamento indesejado, em que o agente girava em torno de si mesmo com alta velocidade angular. Embora esse comportamento permitisse ao agente "trapacear" no aprendizado, colidindo com a bola e enviando-a rapidamente para um lado do campo, não era o objetivo do trabalho. Para lidar com isso, foi estabelecida uma distância mínima inicial de 15 cm entre o robô e a bola na primeira época de treinamento. A cada época, essa distância era aumentada em 10 cm até atingir o limite de 1 metro e meio.

Uma abordagem utilizada por (MEDEIROS; MARCOS; YONEYAMA, 2020) para definir tarefas difíceis no contexto de *Curriculum Learning* foi treinar o agente com a bola em movimento. No entanto, essa abordagem foi descartada no presente trabalho, pois teria um grande impacto na recompensa obtida pelo agente, tornando difícil para o agente determinar se a recompensa obtida foi resultante das ações realizadas por ele ou do ruído na velocidade da bola. Embora essa abordagem seja utilizada em treinamentos com múltiplos agentes, em que a equipe adversária interage com a bola, isso estava fora do escopo do presente trabalho.

O primeiro agente que apresentou sinais de aprendizado foi uma modificação da arquitetura do DDPG rel que não recebia informações sobre a velocidade da bola. Esse agente era capaz de entender que precisava se aproximar da bola e tentar empurrá-la para o gol. No entanto, a falta de informações sobre a velocidade da bola fazia com que o agente não soubesse se a bola estava em movimento ou parada. Isso resultava em erros na maioria das vezes em que o agente tentava chutar a bola. Quando a bola estava parada, o robô chegava um pouco à frente dela e a ultrapassava. Quando a bola estava em movimento, o robô chegava na posição em que a bola estava anteriormente.

4.1 Esquecimento catastrófico nos agentes propostos

Como não é possível a obtenção de um agente capaz de interceptar a bola em movimento nessa abordagem, esse agente, que não recebia informações sobre a velocidade da bola, foi descartado.

Os resultados do treinamento dos modelos TD rel, DDPG abs, TD3 abs e DDPG rel estão ilustrados na Figura 4.26.

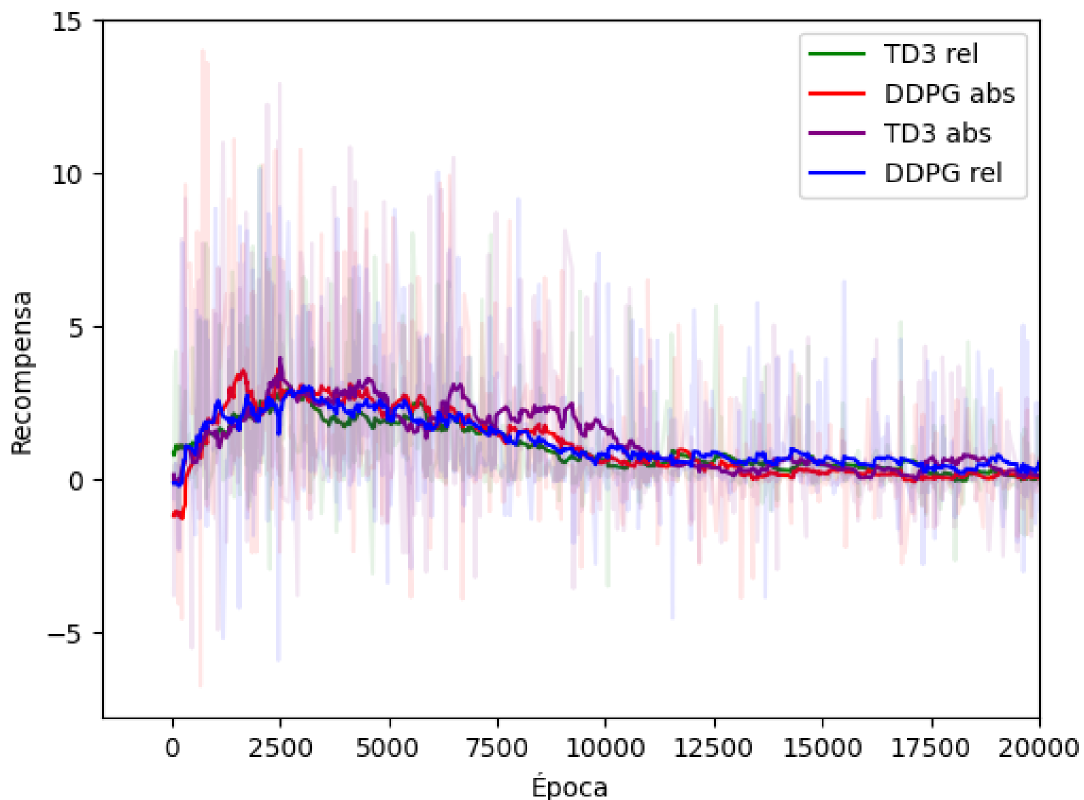

Figura 4.26 – Recompensa ao longo do treinamento com suavização usando média móvel com fator 0.95

Produzido pelo autor

É possível observar que todos os algoritmos apresentam um fenômeno conhecido como esquecimento catastrófico, mesmo o TD3, que segundo a literatura deveria ser mais robusto a esse problema (FUJIMOTO; HOOF; MEGER, 2018). Esse declínio na recompensa média obtida pelo robô é refletido na redução de sua velocidade angular. O robô passa a priorizar apenas se aproximar da bola e ficar parado próximo a ela, com pequenas oscilações na velocidade linear, como mostrado na Figura 4.27.

Esse comportamento pode estar relacionado ao fato de que, na maioria do tempo, o robô precisa se mover em linha reta para alcançar a bola, realizando apenas breves momentos

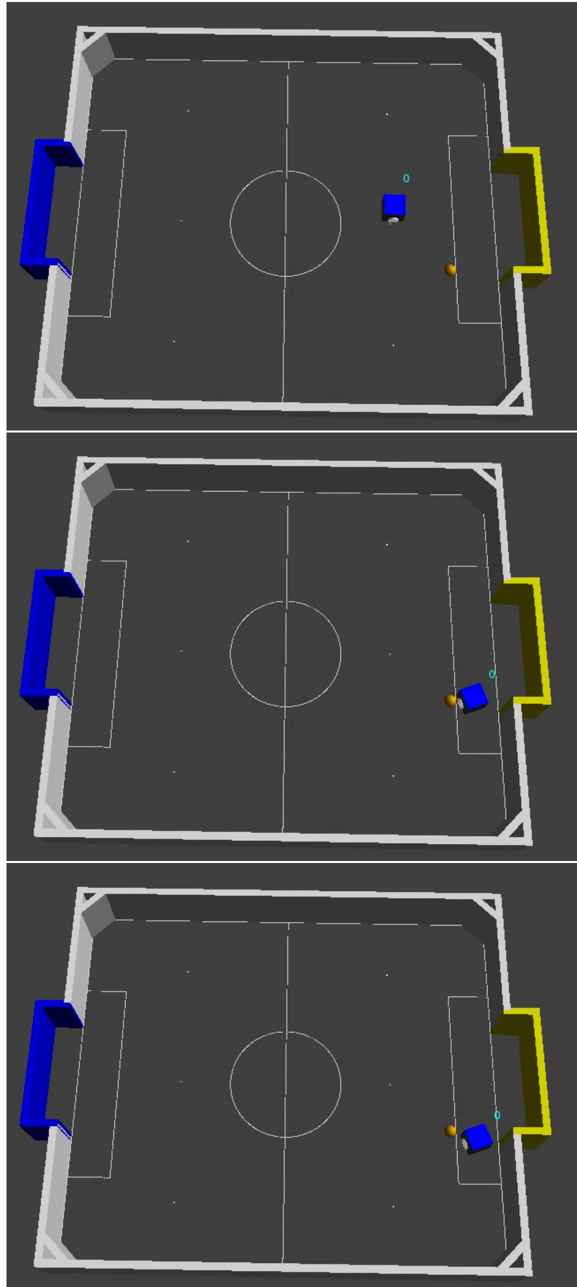

Figura 4.27 – Sequência do jogador após a ocorrência de esquecimento catastrófico. Mantém o aprendizado de como ir para a bola, mas fica parado ao lado dela.

de rotação para se alinhar atrás dela. Esse desequilíbrio entre a quantidade de velocidades lineares e angulares diferentes de zero é agravado pelo fato de que a maior recompensa é obtida quando o robô se move em linha reta e empurra a bola em direção ao gol.

Uma abordagem que pode ser eficaz para lidar com essa questão é uma adaptação inspirada na técnica *Prioritized Experience Replay* (PER), onde, ao invés de priorizar as recompensas obtidas, prioriza-se a mesma quantidade de amostras em que velocidades angulares e lineares diferentes de zero sejam utilizadas durante a etapa de treinamento da rede neural.

Com base nas sugestões de (JEDLICKA et al., 2022) para lidar com o problema de esquecimento catastrófico, realizamos um novo treinamento com uma redução na taxa de aprendizado de 3×10^{-4} para 3×10^{-5}. Essa redução permitiu que a rede ajustasse seus pesos de forma mais refinada. No entanto, a fim de evitar o longo tempo de treinamento necessário até a época 5000 para alcançar os pontos de máximo local observados anteriormente na Figura 4.26, utilizamos os pesos das redes mostradas na Figura 4.26 como ponto de partida para o treinamento das redes mostradas na Figura 4.28. No entanto, mesmo com essas modificações, o problema de esquecimento catastrófico ainda persistiu.

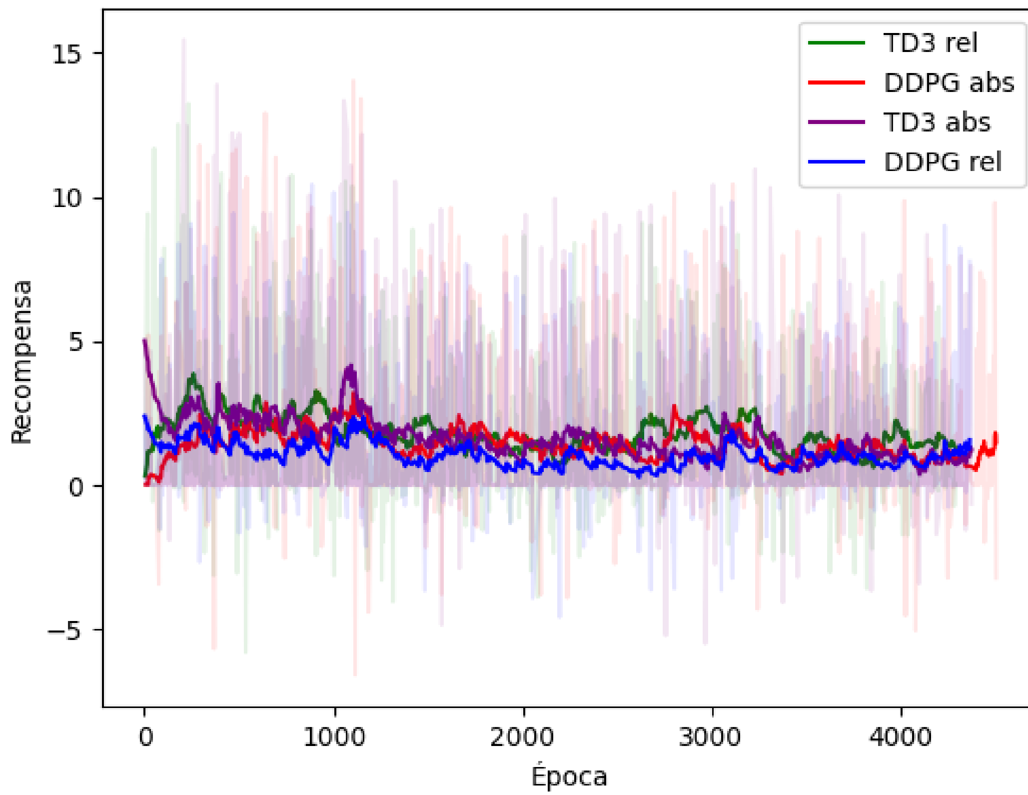

Figura 4.28 – Recompensa ao longo do treinamento com suavização usando média móvel com fator 0.95

Produzido pelo autor

4.2 Utilização de políticas ótimas locais

Uma vez que os algoritmos apresentados na Figura 4.26 alcançaram políticas ótimas locais μ até a época 5000, a resolução do problema de esquecimento catastrófico e a busca por uma política ótima global foram descartadas. Em vez disso, os modelos exibidos na Figura Figura 4.26 foram utilizados para a obtenção de resultados numéricos.

Mesmo sem atingir uma política ótima, o algoritmo TD3 abs demonstra a habilidade de interceptar a bola em movimento, como ilustrado na sequência de imagens da Figura 4.29.

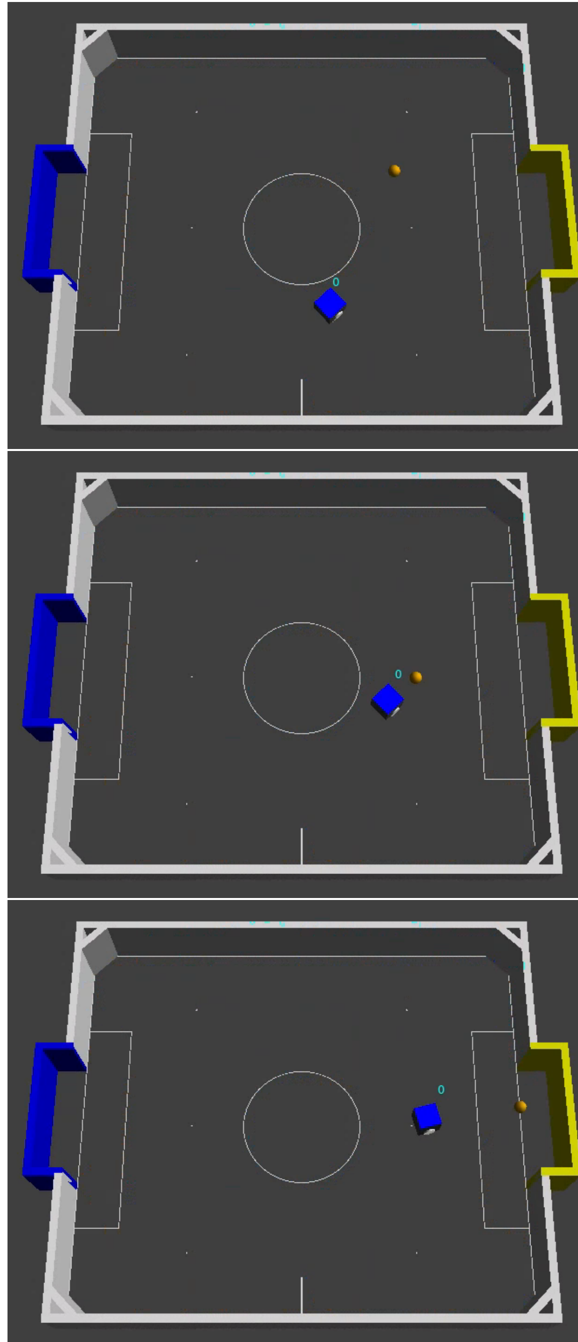

Figura 4.29 – Sequência com a bola em movimento. O robô se desloca para uma posição que possibilita a interceptação da bola e a realização do gol.

Além disso, a Figura 4.30 mostra a capacidade aprendida pelo agente de posicionar-se atrás da bola para direcioná-la em direção ao gol. Esses comportamentos foram observados nos 4 agentes treinados.

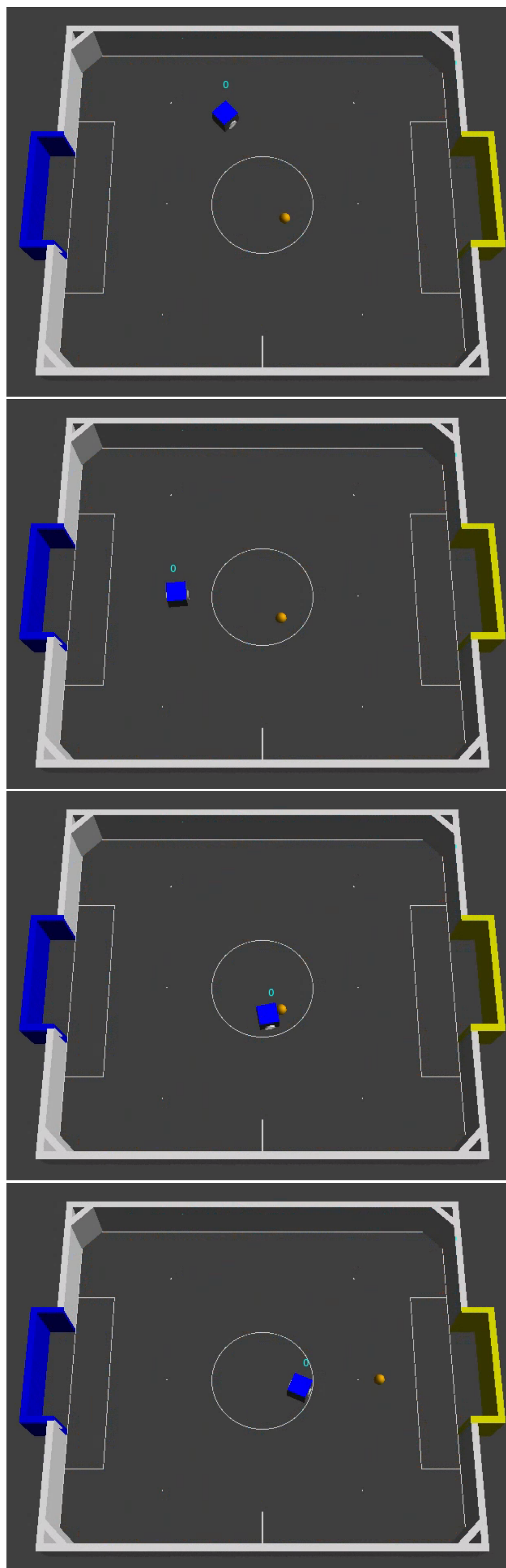

Figura 4.30 – Sequência do jogador chegando atrás da bola de forma a ficar alinhado para “chutar” a bola ao gol adversário.

4.3 Análise quantitativa da desempenho dos agentes

Embora os gráficos de recompensa sejam comumente usados para avaliar o desempenho dos agentes treinados com algoritmos de aprendizado por reforço, eles têm a limitação de depender da mesma função de recompensa para todos os trabalhos analisados. Além disso, a abordagem tradicional de adicionar ruído às ações dos agentes para promover a exploração também interfere nesse método de coleta de resultados.

Visando avaliar a desempenho de agentes, o presente trabalho utiliza uma análise quantitativa para mensurar a capacidade de cada agente em realizar a tarefa de aproximar a bola até o gol adversário, em comparação com o algoritmo utilizado pela equipe UnBall. A equipe UnBall utiliza uma máquina de estados para determinar o campo vetorial a ser gerado com base no estado do jogo. Esse campo vetorial é então usado para calcular o ângulo desejado, permitindo que o robô siga o campo e se mova em direção à bola, conforme ilustrado na Figura 2.21.

Para obter os resultados apresentados na Tabela 4.1, a bola é sempre iniciada no centro do campo. O robô é inicializado em uma posição aleatória no campo e é medida a distância mínima entre a bola e o gol. Esse processo de coleta de dados é repetido por mil épocas, com duração de 20 segundos cada. O tempo gasto para realizar a ação também é registrado na tabela, porém, é importante destacar que a análise do tempo não é um indicador robusto em casos em que o agente fica preso na parede ou não consegue marcar um gol. É válido ressaltar que a semente 747 foi utilizada para gerar a posição aleatória inicial dos robôs, garantindo que os algoritmos analisados iniciem nas mesmas posições.

Tabela 4.1 – Distância da bola ao gol adversário.

Algoritmo	Distância mínima (m)	Tempo gasto (s)
TD3 rel	0.308 ± 0.260	8.908 ± 6.115
DDPG abs	0.309 ± 0.292	8.630 ± 6.367
TD3 abs	0.265 ± 0.234	9.737 ± 5.451
DDPG rel	0.208 ± 0.236	9.868 ± 5.769
UnBall	0.0188 ± 0.105	3.067 ± 1.523

Fonte: Produzido pelos autor.

Nota: Valores menores indicam melhores resultados.

Para uma análise mais detalhada das informações apresentadas na Tabela 4.1, foram gerados os gráficos mostrados na Figura 4.31 e Figura 4.32. Observa-se que, embora nenhuma das quatro redes neurais tenha alcançado um desempenho equivalente ao algoritmo utilizado pela equipe UnBall, todas apresentam baixos valores de distância mínima até o gol. No entanto, não é possível chegar a uma conclusão analisando apenas os valores relacionados ao tempo.

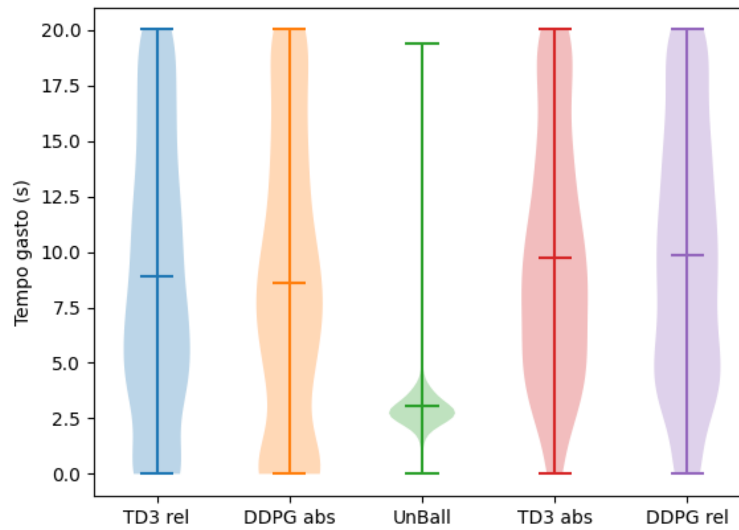

Figura 4.31 – Tempo gasto para obter a menor distância da bola ao gol. Violinos com informação da média ao centro. Valores menores indicam melhores resultados.

Produzido pelo autor

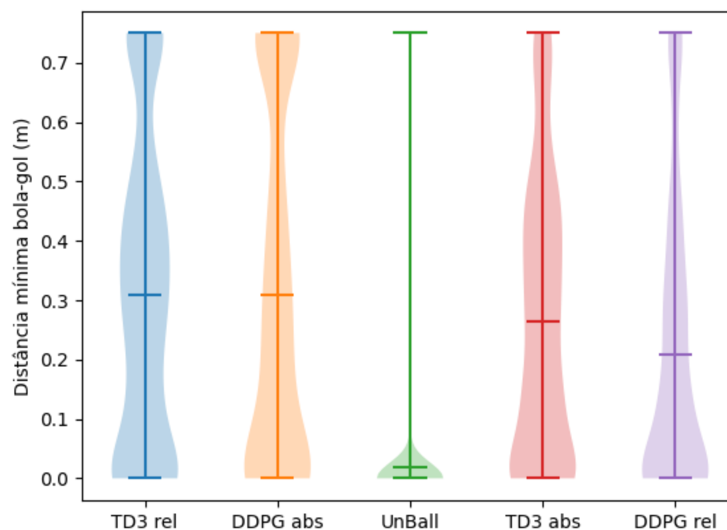

Figura 4.32 – Menor distância obtida da bola em relação ao gol. Violinos com informação da média ao centro. Valores menores indicam melhores resultados.

Produzido pelo autor

O resultado apresentado na Figura 4.32 é semelhante ao obtido por (KALEJAIYE, 2019) ao utilizar o algoritmo PPO. A utilização de vetores relativos permite que a rede neural obtenha um melhor desempenho, especialmente em situações em que é necessário manter

velocidades angulares e lineares constantes, como ao se mover em linha reta em direção ao gol adversário. Por outro lado, ao utilizar vetores absolutos como entrada, a variação constante na entrada dificulta a manutenção de uma saída constante.

4.4 Análise qualitativa do desempenho dos agentes

Foi realizado um experimento semelhante ao descrito em (MARTINS et al., 2021), no qual foram realizadas duas partidas[1] de 10 minutos entre o melhor agente treinado por aprendizado por reforço e o agente da equipe UnBall no modo 1 contra 1. Para isso, utilizou-se o algoritmo DDPG rel. No primeiro jogo, a equipe UnBall venceu por 24 a 20, enquanto no segundo jogo o DDPG rel venceu por 30 a 18. Embora os placares possam contradizer os resultados anteriores, a vitória da rede neural ocorreu devido ao sistema da UnBall se afastar e abrir espaço para a rede neural marcar gols diretamente.

Na partida 1 contra 1, foi observado que, caso o robô da UnBall fique preso na parede, sua máquina de estados faz com que ele dê ré e se vire. Essa abordagem não está presente na rede neural. É importante ressaltar que cada estado possível na máquina de estados da UnBall precisou ser implementado e ajustado manualmente.

4.5 Considerações sobre os resultados obtidos

É importante ressaltar que, à medida que a estratégia da equipe UnBall se torna mais robusta, o número de linhas de código aumenta, o que dificulta a manutenção do sistema e a correção de problemas. Embora as redes neurais apresentadas neste trabalho não tenham superado as métricas alcançadas por uma estratégia de máquina de estados com controle, a simplicidade do sistema e a possibilidade de um novo método de controle do robô tornam a aplicação de redes neurais uma opção a ser considerada.

[1] O vídeo da partida está disponível em: https://youtu.be/sJnSp8uFxP4

5 Conclusões

Esta monografia abordou o problema de desenvolver um agente autônomo para jogar futebol de robôs na categoria *Very Small Size Soccer* (VSSS). Foi proposto o uso de aprendizado por reforço através dos algoritmos *Twin Delayed DDPG* (TD3) e *Deep Deterministic Policy Gradient* (DDPG) sob a motivação de substituir uma máquina de estados para definir o comportamento do agente. A utilização do aprendizado por reforço eliminou a necessidade de um bloco de controle para guiar o agente com lógica pré-definida, evitando a necessidade de ajustes constantes.

Com base nos resultados apresentados neste trabalho, é possível responder à hipótese inicialmente proposta: sim, é possível utilizar a rede neural proposta para para criar um robô autônomo capaz de jogar futebol. Além disso, foi demonstrado que o agente treinado pode competir com algoritmos baseados na arquitetura convencional de máquina de estados e controle.

Embora o agente de aprendizado por reforço tenha se aproveitado da abertura de frente ao gol deixada pelo robô da equipe de competição UnBall, que utiliza máquina de estados, a partida teve momentos em que ambos necessitaram disputar a posse da bola e realizar as tarefas de interceptação da bola em movimento, direcioná-la em direção ao gol adversário e impedir que a bola entrasse no próprio gol. É importante considerar que o projeto atual da equipe UnBall foi desenvolvido ao longo de 4 anos, enquanto o resultado obtido pelo agente de aprendizado por reforço foi alcançado em 1 ano e meio.

Para trabalhos futuros, pretende-se explorar a utilização de ruído nos parâmetros da rede ator em vez de na saída da ação, conforme proposto por *Adaptive Parameter Noise* (PLAPPERT et al., 2017), para obter uma convergência mais rápida no aprendizado da rede neural. Também é desejável a aplicação do algoritmo *Multi-Agent DDPG* (MADDPG) (LOWE et al., 2017), uma vez que este trabalho se concentrou em um único agente, mas a tarefa de futebol de robôs requer cooperação entre múltiplos agentes para ser efetiva.

Os algoritmos *model-based* estão alcançando desempenho comparável aos algoritmos *model-free*, como o *Dreamer* proposto por (HAFNER et al., 2019). Uma análise da aplicabilidade desses algoritmos no contexto do futebol de robôs também é desejada, considerando que o *Dreamer* apresenta um tempo de aprendizado mais curto do que os algoritmos *model-free* de última geração em diversos ambientes de aprendizado por reforço.

Referências

ABIODUN, O. I.; JANTAN, A.; OMOLARA, A. E.; DADA, K. V.; UMAR, A. M.; LINUS, O. U.; ARSHAD, H.; KAZAURE, A. A.; GANA, U.; KIRU, M. U. Comprehensive review of artificial neural network applications to pattern recognition. **IEEE access**, IEEE, v. 7, p. 158820–158846, 2019. Citado na p. 16.

ANDREA ASPERTI. **a didactic environment for Deep Reinforcement Learning - Scientific Figure on ResearchGate**. Disponível em: https://www.researchgate.net/figure/Examples-of-catastrophic-forgetting-during-training-of-DDPG-left-and-DSAC-right_fig2_362174884 – acesso em 6 jun. 2023. 2008. Citado na p. 32.

ANDRYCHOWICZ, M.; RAICHUK, A.; STAŃCZYK, P.; ORSINI, M.; GIRGIN, S.; MARINIER, R.; HUSSENOT, L.; GEIST, M.; PIETQUIN, O.; MICHALSKI, M. et al. What matters in on-policy reinforcement learning? a large-scale empirical study. **arXiv preprint arXiv:2006.05990**, 2020. Citado na p. 50.

BALAMURUGAN, E.; MEHBODNIYA, A.; KARIRI, E.; YADAV, K.; KUMAR, A.; HAQ, M. A. Network optimization using defender system in cloud computing security based intrusion detection system withgame theory deep neural network (IDSGT-DNN). **Pattern Recognition Letters**, Elsevier, v. 156, p. 142–151, 2022. Citado na p. 14.

BASSANI, H. F.; DELGADO, R. A.; JUNIOR, J. N. d. O. L.; MEDEIROS, H. R.; BRAGA, P. H.; MACHADO, M. G.; SANTOS, L. H.; TAPP, A. A framework for studying reinforcement learning and sim-to-real in robot soccer. **arXiv preprint arXiv:2008.12624**, 2020. Citado nas pp. 33, 38, 39.

BHATT, A.; ARGUS, M.; AMIRANASHVILI, A.; BROX, T. Crossnorm: Normalization for off-policy td reinforcement learning. **arXiv preprint arXiv:1902.05605**, 2019. Citado na p. 49.

BURKART, N.; HUBER, M. F. A survey on the explainability of supervised machine learning. **Journal of Artificial Intelligence Research**, v. 70, p. 245–317, 2021. Citado na p. 16.

CALDARINI, G.; JAF, S.; MCGARRY, K. A literature survey of recent advances in chatbots. **Information**, MDPI, v. 13, n. 1, p. 41, 2022. Citado na p. 14.

DUBEY, S. R.; SINGH, S. K.; CHAUDHURI, B. B. Activation functions in deep learning: A comprehensive survey and benchmark. **Neurocomputing**, Elsevier, 2022. Citado nas pp. 17, 51.

ELALLID, B. B.; BENAMAR, N.; HAFID, A. S.; RACHIDI, T.; MRANI, N. A comprehensive survey on the application of deep and reinforcement learning approaches in autonomous driving. **Journal of King Saud University-Computer and Information Sciences**, Elsevier, v. 34, n. 9, p. 7366–7390, 2022. Citado na p. 14.

FUJIMOTO, S.; HOOF, H.; MEGER, D. Addressing function approximation error in actor-critic methods. In: PMLR. INTERNATIONAL conference on machine learning. 2018. P. 1587–1596. Citado nas pp. 31, 53, 56.

GOMINSU. **File:** .png. Disponível em: `https:https://wikidocs.net/175460` – acesso em 22 jun. 2023. 2008. Citado na p. 31.

GRONDMAN, I.; BUSONIU, L.; LOPES, G. A.; BABUSKA, R. A survey of actor-critic reinforcement learning: Standard and natural policy gradients. **IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)**, IEEE, v. 42, n. 6, p. 1291–1307, 2012. Citado na p. 28.

HAFNER, D.; LILLICRAP, T.; BA, J.; NOROUZI, M. Dream to control: Learning behaviors by latent imagination. **arXiv preprint arXiv:1912.01603**, 2019. Citado na p. 65.

HELM, J. M.; SWIERGOSZ, A. M.; HAEBERLE, H. S.; KARNUTA, J. M.; SCHAFFER, J. L.; KREBS, V. E.; SPITZER, A. I.; RAMKUMAR, P. N. Machine learning and artificial intelligence: definitions, applications, and future directions. **Current reviews in musculoskeletal medicine**, Springer, v. 13, p. 69–76, 2020. Citado na p. 16.

HOU, Y.; ZHANG, Y. Improving DDPG via prioritized experience replay. **no. May**, 2019. Citado na p. 49.

IOFFE, S.; SZEGEDY, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In: PMLR. INTERNATIONAL conference on machine learning. 2015. P. 448–456. Citado na p. 49.

JEDLICKA, P.; TOMKO, M.; ROBINS, A.; ABRAHAM, W. C. Contributions by metaplasticity to solving the Catastrophic Forgetting Problem. **Trends in Neurosciences**, Elsevier, 2022. Citado na p. 58.

JUAN, N. P.; VALDECANTOS, V. N. Review of the application of Artificial Neural Networks in ocean engineering. **Ocean Engineering**, Elsevier, v. 259, p. 111947, 2022. Citado na p. 17.

KAISER, L.; BABAEIZADEH, M.; MILOS, P.; OSINSKI, B.; CAMPBELL, R. H.; CZECHOWSKI, K.; ERHAN, D.; FINN, C.; KOZAKOWSKI, P.; LEVINE, S. et al. Model-based reinforcement learning for atari. **arXiv preprint arXiv:1903.00374**, 2019. Citado na p. 21.

KALEJAIYE, L. B. T. **Treinamento de agentes jogadores de futebol usando aprendizado por reforço**. 2019. F. 51. Monografia (Graduação) – Universidade de Brasília, Brasília. Disponível em: <https://bdm.unb.br/handle/10483/28124>. Citado nas pp. 38, 39, 42, 43, 45, 63.

KAUR, R.; GHOLAMHOSSEINI, H.; SINHA, R.; LINDÉN, M. Melanoma classification using a novel deep convolutional neural network with dermoscopic images. **Sensors**, MDPI, v. 22, n. 3, p. 1134, 2022. Citado na p. 14.

KIM, J.-H.; KIM, Y.-J.; KIM, D.-H.; SEOW, K.-T. 1. Soccer Robotics. In: SOCCER Robotics. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. P. 1–26. ISBN 978-3-540-40921-2. DOI: 10.1007/978-3-540-40921-2_1. Disponível em: <https://doi.org/10.1007/978-3-540-40921-2_1>. Citado nas pp. 15, 33–37, 39, 40, 42, 43, 48.

LILLICRAP, T. P.; HUNT, J. J.; PRITZEL, A.; HEESS, N.; EREZ, T.; TASSA, Y.; SILVER, D.; WIERSTRA, D. Continuous control with deep reinforcement learning. **arXiv preprint arXiv:1509.02971**, 2015. Citado nas pp. 28–30, 53.

LIM, Y.; CHOI, S.-H.; KIM, J.-H.; KIM, D.-H. Evolutionary univector field-based navigation with collision avoidance for mobile robot. **IFAC Proceedings Volumes**, Elsevier, v. 41, n. 2, p. 12787–12792, 2008. Citado nas pp. 41, 43.

LOWE, R.; WU, Y. I.; TAMAR, A.; HARB, J.; PIETER ABBEEL, O.; MORDATCH, I. Multi-agent actor-critic for mixed cooperative-competitive environments. **Advances in neural information processing systems**, v. 30, 2017. Citado na p. 65.

MAKAROV, P. A.; YIRTICI, T.; AKKAYA, N.; AYTAC, E.; SAY, G.; BURGE, G.; YILMAZ, B.; ABIYEV, R. H. A Model-Free Algorithm of Moving Ball Interception by Holonomic Robot Using Geometric Approach. In: CHALUP, S.; NIEMUELLER, T.; SUTHAKORN, J.; WILLIAMS, M.-A. (Ed.). **RoboCup 2019: Robot World Cup XXIII**. Cham: Springer International Publishing, 2019. P. 166–175. ISBN 978-3-030-35699-6. Citado na p. 41.

MARTINS, F. B.; MACHADO, M. G.; BASSANI, H. F.; BRAGA, P. H. M.; BARROS, E. S. **rSoccer: A Framework for Studying Reinforcement Learning in Small and Very Small Size Robot Soccer**. arXiv, 2021. DOI: 10.48550/ARXIV.2106.12895. Disponível em: <https://arxiv.org/abs/2106.12895>. Citado nas pp. 14, 38, 39, 47, 54, 64.

MEDEIROS, T. F. de; MARCOS, R. d. A.; YONEYAMA, T. Deep Reinforcement Learning Applied to IEEE Very Small Size Soccer Strategy. In: IEEE. 2020 Latin American Robotics Symposium (LARS), 2020 Brazilian Symposium on Robotics (SBR) and 2020 Workshop on Robotics in Education (WRE). 2020. P. 1–6. Citado nas pp. 41, 55.

MISRA, D. Mish: A self regularized non-monotonic activation function. **arXiv preprint arXiv:1908.08681**, 2019. Citado na p. 51.

MNIH, V.; KAVUKCUOGLU, K.; SILVER, D.; GRAVES, A.; ANTONOGLOU, I.; WIERSTRA, D.; RIEDMILLER, M. Playing atari with deep reinforcement learning. **arXiv preprint arXiv:1312.5602**, 2013. Citado nas pp. 23, 24.

MORALES, M. **Grokking deep reinforcement learning**. Manning Publications, 2020. Citado nas pp. 24, 25.

MURDOCH, W. J.; SINGH, C.; KUMBIER, K.; ABBASI-ASL, R.; YU, B. Interpretable machine learning: definitions, methods, and applications. **arXiv preprint arXiv:1901.04592**, 2019. Citado na p. 16.

OUYANG, L.; WU, J.; JIANG, X.; ALMEIDA, D.; WAINWRIGHT, C.; MISHKIN, P.; ZHANG, C.; AGARWAL, S.; SLAMA, K.; RAY, A. et al. Training language models to follow instructions with human feedback. **Advances in Neural Information Processing Systems**, v. 35, p. 27730–27744, 2022. Citado na p. 14.

PATURI, U. M. R.; CHERUKU, S.; REDDY, N. The role of artificial neural networks in prediction of mechanical and tribological properties of composites—a comprehensive review. **Archives of Computational Methods in Engineering**, Springer, v. 29, n. 5, p. 3109–3149, 2022. Citado na p. 17.

PENA MATEUS MACHADO, M. B. carlos. **An analysis of Reinforcement Learning applied to Coach task in IEEE Very Small Size Soccer - Scientific Figure on ResearchGate.** Disponível em: https://www.researchgate.net/figure/IEEE-Very-Small-Size-Soccer_fig1_346302409 – acesso em 27 jan. 2023. 2020. Citado nas pp. 33, 35, 38.

PLAPPERT, M.; HOUTHOOFT, R.; DHARIWAL, P.; SIDOR, S.; CHEN, R. Y.; CHEN, X.; ASFOUR, T.; ABBEEL, P.; ANDRYCHOWICZ, M. Parameter space noise for exploration. **arXiv preprint arXiv:1706.01905**, 2017. Citado na p. 65.

RIJSDIJK, J.; WU, L.; PERIN, G.; PICEK, S. Reinforcement learning for hyperparameter tuning in deep learning-based side-channel analysis. **IACR Transactions on Cryptographic Hardware and Embedded Systems**, p. 677–707, 2021. Citado na p. 17.

SAHU, S. K.; MOKHADE, A.; BOKDE, N. D. An Overview of Machine Learning, Deep Learning, and Reinforcement Learning-Based Techniques in Quantitative Finance: Recent Progress and Challenges. **Applied Sciences**, MDPI, v. 13, n. 3, p. 1956, 2023. Citado na p. 16.

SHRIDHAR, A.; TOMSON, P.; INNES, M. Interoperating deep learning models with onnx. jl. In: 1. PROCEEDINGS of the JuliaCon Conferences. 2020. v. 1, p. 59. Citado na p. 53.

SILVER, D.; LEVER, G.; HEESS, N.; DEGRIS, T.; WIERSTRA, D.; RIEDMILLER, M. Deterministic policy gradient algorithms. In: PMLR. INTERNATIONAL conference on machine learning. 2014. P. 387–395. Citado na p. 29.

SOVIANY, P.; IONESCU, R. T.; ROTA, P.; SEBE, N. Curriculum learning: A survey. **International Journal of Computer Vision**, Springer, p. 1–40, 2022. Citado nas pp. 41, 50.

SUTTON, R. S.; MCALLESTER, D.; SINGH, S.; MANSOUR, Y. Policy gradient methods for reinforcement learning with function approximation. **Advances in neural information processing systems**, v. 12, 1999. Citado nas pp. 25–27.

SUTTON, R. S.; BARTO, A. G. Reinforcement Learning: An Introduction. In: REINFORCEMENT Learning: An Introduction. London, England: The MIT Press, 2014. P. 1–264. Disponível em: <https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf>. Citado nas pp. 18–21.

TU, K.-Y. Design and Implementation of a Soccer Robot with Modularized Control Circuits. In: BIRK, A.; CORADESCHI, S.; TADOKORO, S. (Ed.). **RoboCup 2001: Robot Soccer World Cup V**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002. P. 459–464. ISBN 978-3-540-45603-2. Citado na p. 41.

ULLO, S. L. **Fig. 1. Q-Learning vs. Deep Q-Learning**. Disponível em: https://https://www.researchgate.net/figure/Q-Learning-vs-Deep-Q-Learning_fig1_351884746 – acesso em 11 fev. 2023. 2021. Citado na p. 23.

USAMA, M.; QADIR, J.; RAZA, A.; ARIF, H.; YAU, K.-L. A.; ELKHATIB, Y.; HUSSAIN, A.; AL-FUQAHA, A. Unsupervised machine learning for networking: Techniques, applications and research challenges. **IEEE access**, IEEE, v. 7, p. 65579–65615, 2019. Citado na p. 16.

VAN HASSELT, H.; GUEZ, A.; SILVER, D. Deep reinforcement learning with double q-learning. In: 1. PROCEEDINGS of the AAAI conference on artificial intelligence. 2016. v. 30. Citado nas pp. 24, 30.

WATKINS, C. J.; DAYAN, P. Q-learning. **Machine learning**, Springer, v. 8, p. 279–292, 1992. Citado nas pp. 21, 22.

WIKIPÉDIA. **Função sigmoide — Wikipédia, a enciclopédia livre**. 2021. [Online; accessed 25-maio-2021]. Disponível em: <https://pt.wikipedia.org/w/index.php?title=Fun%C3%A7%C3%A3o_sigmoide&oldid=61241436>. Citado na p. 17.

WIKIPÉDIA. **Tangente hiperbólica — Wikipédia, a enciclopédia livre**. [Online; accessed 22-janeiro-2023]. 2023. Disponível em: <%5Curl%7Bhttps://pt.wikipedia.org/w/index.php?title=Tangente_hiperb%C3%B3lica&oldid=65155230%7D>. Citado na p. 18.

ZHANG, J.; HE, T.; SRA, S.; JADBABAIE, A. Why gradient clipping accelerates training: A theoretical justification for adaptivity. **arXiv preprint arXiv:1905.11881**, 2019. Citado na p. 50.

ZHANG, S.; SUTTON, R. S. A deeper look at experience replay. **arXiv preprint arXiv:1712.01275**, 2017. Citado na p. 47.

Apêndices

Apêndice A – Códigos de programação

A.1 Classe de comunicação entre o FiraSIM e o agente

Código A.1 – Código de Python

```
import time
import random
import math
import numpy as np
from firasim_client.libs.kdtree import KDTree
from firasim_client.libs.firasim import (FIRASimCommand,
    FIRASimVision)
from firasim_client.libs.entities import Field
from firasim_client.libs.entities import (Robot, Ball, NUM_ALLIES,
    NUM_OPPONENTS)


# IMPORTANT:
# REWARDS ARE USING JUST INFO ABOUT ROBOT 0. NOT ABOUT ALL ALLY
    ROBOTS
# ALLY GOAL SUPOSED TO BEE ALWAYES IN NEGATIVE X

def x():
    return random.uniform(-Field.width/2 + 0.10, Field.width/2 -
        0.10)

def y():
    return random.uniform(-Field.height/2 + 0.10, Field.height/2 -
        0.10)

def theta():
    return random.uniform(0, 2*math.pi)



class MarkovDecisionProcess:
    def __init__(self,
                 max_v: float,
                 max_w: float,
                 max_time_per_episode: int = 60*5,
                 team_color: str = "blue",
                 opponent_color: str = "yellow",
                 num_allies_in_field: int = 1,
                 num_opponents_in_field: int = 0,
```

```
35                     time_step: float = 0.02,
36                     min_dist: float = 0.10,
37                     max_dist: float = 0.15,
38                     ):
39 
40         self.min_dist = min_dist
41         self.max_dist = max_dist
42 
43         self.max_v = max_v
44         self.max_w = max_w
45 
46         self.num_allies_in_field = num_allies_in_field
47         self.num_opponents_in_field = num_opponents_in_field
48 
49         self.vision = FIRASimVision()
50         self.ally_command = FIRASimCommand( team_yellow = True
51                                             if team_color ==
                                                   "yellow"
52                                             else False
53                                             )
54 
55         self.current_step = 0
56         self.max_time_per_episode = max_time_per_episode
57         self.time_step = time_step
58 
59         self.ball = Ball()
60         self.ally_robots = [Robot(id=id, ally=True,
               color=team_color)
61                             for id in range(NUM_ALLIES)]
62         self.opponent_robots = [Robot(id=id, ally=False,
               color=opponent_color)
63                                 for id in range(NUM_OPPONENTS)]
64         self.field = Field(team_color=team_color)
65 
66         self.previous_ball_potential = None
67 
68     def step(self, actions):
69 
70         self.send_velocities2firasim(actions)
71         firasim_frame = self.get_firasim_frame()
72         self.update_entity_properties(firasim_frame)
73         self.current_step += 1
74 
75         return (self.process_state(), self.reward(), self.done())
76 
77     def send_velocities2firasim(self, actions):
78         """actions: tuple [[V, W]]"""
79 
80         tmp_actions = []
81         for id in range(self.num_allies_in_field):
82             vels = Robot.vel_vw2vel_whells(*actions[id])
83             vl, vr = vels["vl"], vels["vr"]
```

```
84                 tmp_actions.append((vl, vr))
85 
86 
87             self.ally_command.writeMulti(tmp_actions)
88 
89         def get_firasim_frame(self):
90             last_packet = None
91 
92             last_time = time.time()
93             while True:
94                 if (time.time() - last_time) < self.time_step:
95                     packet = self.vision.read()
96                     if packet is not None:
97                         last_packet = packet
98                 else:
99                     if last_packet == None:
100                        last_time = time.time()
101                    else:
102                        break
103 
104            return last_packet
105 
106        def update_entity_properties(self, message):
107 
108            if self.field.team_color == "blue":
109                robots = message.frame.robots_blue
110            elif self.field.team_color == "yellow":
111                robots = message.frame.robots_yellow
112 
113            for robot in robots:
114                self.ally_robots[robot.robot_id].pos = [robot.x,
                      robot.y, robot.orientation]
115                self.ally_robots[robot.robot_id].velxy = [robot.vx,
                      robot.vy]
116                self.ally_robots[robot.robot_id].w = robot.vorientation
117 
118            self.ball.pos = [message.frame.ball.x,
                  message.frame.ball.y]
119            self.ball.velxy = [message.frame.ball.vx,
                  message.frame.ball.vy]
120 
121        def process_state(self):
122            state = {'ally': {},
123                     'opponent': {},
124                     'ball': {},
125                     }
126 
127            for id in range(self.num_allies_in_field):
128                state['ally'][id] = {
129                    'pos_xy':
130                        np.array([self.ally_robots[id].pos[0],
131                                  self.ally_robots[id].pos[1]]),
```

```
                'theta': self.ally_robots[id].pos[2],
                'vel_xy': np.array(self.ally_robots[id].velxy),
                'w': self.ally_robots[id].w,
                                }

        for id in range(self.num_opponents_in_field):
            state['opponent'][id] = {
                'pos_xy':
                    np.array([self.opponent_robots[id].pos[0],
                            self.opponent_robots[id].pos[1]]),
                'theta': self.opponent_robots[id].pos[2],
                'vel_xy': np.array(self.opponent_robots[id].velxy),
                'w': self.opponent_robots[id].w,
                                    }

        state['ball'] = {'pos_xy':np.array(self.ball.pos),
                         'vel_xy': np.array(self.ball.velxy),
                         }

        return state

    def reset_random_init_pos(self, fixed_ball_pos=False):
        self.init_time = time.time()

        self.set_entities_positions(fixed_ball_pos)

        self.send_positions2fira()

        return self.process_state()

    def set_entities_positions(self, fixed_ball_pos):
        if fixed_ball_pos:
            ball_pos = [0,0]
        else:
            ball_pos = [x(), y()]

        init_pos = {"ball": ball_pos,
                    "allies": [],
                    "opponents": [],
                    }

        places = KDTree()
        places.insert(ball_pos)

        for id in range(self.num_allies_in_field):
            if id == 0:
                continue
            pos = [x(), y()]
            while places.get_nearest(pos)[1] < self.min_dist or
               places.get_nearest(pos)[1] > self.max_dist:
                pos = [x(), y()]

```

```
183             places.insert(pos)
184             init_pos["allies"].append(pos)
185 
186         if self.num_allies_in_field < NUM_ALLIES:
187             for id in range(self.num_allies_in_field, NUM_ALLIES):
188                 if id != 0:
189                     self.ally_robots[id].move_outside_field()
190 
191         for _ in range(self.num_opponents_in_field):
192             pos = [x(), y()]
193             while places.get_nearest(pos)[1] < self.min_dist:
194                 pos = [x(), y()]
195 
196             places.insert(pos)
197             init_pos["opponents"].append(pos)
198 
199         if self.num_opponents_in_field < NUM_OPPONENTS:
200             for id in range(self.num_opponents_in_field,
                   NUM_OPPONENTS):
201                 if id != 0:
202                     self.opponent_robots[id].move_outside_field()
203 
204         for id in range(self.num_allies_in_field):
205             if id != 0:
206                 self.ally_robots[id].pos =
                        [*init_pos["allies"][id], theta()]
207             else:
208                 self.ally_robots[id].pos = [-0.3,0, 0]
209 
210         for id in range(self.num_opponents_in_field):
211             if id != 0:
212                 self.opponent_robots[id].pos =
                        [*init_pos["opponents"][id], theta()]
213             else:
214                 self.opponent_robots[id].pos = [0.3, 0, 0]
215 
216         self.ball.pos = init_pos["ball"]
217 
218     def send_positions2fira(self):
219         team_yellow = True if self.field.team_color == "yellow"
               else False
220 
221         opponent_command = FIRASimCommand(team_yellow = not
               team_yellow)
222 
223         self.ally_command.setBallPos(self.ball.pos[0],
               self.ball.pos[1])
224 
225         for id in range(NUM_ALLIES):
226             self.ally_command.setPos(id, *self.ally_robots[id].pos)
227 
228         for id in range(NUM_OPPONENTS):
```

```
229             opponent_command.setPos(id,
                   *self.opponent_robots[id].pos)
230 
231     def reward(self):
232         """
233         It's a simplified reward, that give +1 if the agent do a
               goal and
234         return -1 reward if the agent receive a goal
235         """
236         reward = 0
237 
238         # ALLY GOAL SUPOSED TO BEE ALWAYES IN NEGATIVE X
239 
240         if self.ball.pos[0] > self.field.opponent_goal_pos[0]:
241             reward = 1
242         elif self.ball.pos[0] < self.field.ally_goal_pos[0]:
243             reward = -1
244         else:
245 
246             w_move = 0.2
247             w_ball_grad = 0.8
248             w_energy = 2e-4
249 
250             # Calculate ball potential
251             grad_ball_potential = self.ball_grad()
252             # Calculate Move ball
253             move_reward = self.move_reward()
254             # Calculate Energy penalty
255             energy_penalty = self.energy_penalty()
256 
257             reward = (  w_move * move_reward + \
258                         w_ball_grad * grad_ball_potential + \
259                         w_energy * energy_penalty
260                         )
261 
262         return reward
263 
264     def done(self):
265 
266         if (time.time() - self.init_time) >
               self.max_time_per_episode:
267             return 1
268 
269         return 1 if abs(self.ball.pos[0]) >
               abs(self.field.ally_goal_pos[0]) else 0
270 
271     def energy_penalty(self):
272 
273         energy_penalty = 0
274 
275         for id in range(self.num_allies_in_field):
```

```
276             linearVelocity =
                   math.sqrt(self.ally_robots[id].velxy[0]**2 +
                   self.ally_robots[id].velxy[1]**2)
277             en_penalty_1 = abs(linearVelocity)
278             en_penalty_2 = abs(self.ally_robots[id].w)
279             energy_penalty -= (en_penalty_1 + en_penalty_2)
280 
281         return energy_penalty
282 
283     def move_reward(self):
284         '''
285         Cosine between the robot vel vector and the vector robot
               -> ball.
286         This indicates rather the robot is moving towards the ball
               or not.
287         '''
288 
289         ball_pos = np.array(self.ball.pos)
290         robot_pos = np.array([self.ally_robots[0].pos[0],
               self.ally_robots[0].pos[1]])
291         robot_velxy = np.array(self.ally_robots[0].velxy)
292         robot_ball = ball_pos - robot_pos
293         unit_robot_ball = robot_ball/np.linalg.norm(robot_ball)
294 
295         move_reward = np.dot(unit_robot_ball, robot_velxy)
296 
297         move_reward = np.clip(move_reward / Robot.max_velxy_norm,
               -1.0, 1.0)
298         return move_reward
299 
300     def ball_grad(self):
301         '''
302         Cosine between the ball vel vector and the vector ball ->
               goal.
303         This indicates rather the ball is moving towards the goal
               or not.
304         '''
305 
306         ball_pos = np.array(self.ball.pos)
307         opponent_goal_pos = np.array(self.field.opponent_goal_pos)
308         ball_velxy = np.array(self.ball.velxy)
309         ball_opponent_goal = opponent_goal_pos - ball_pos
310         unit_ball_opponent_goal =
               ball_opponent_goal/np.linalg.norm(ball_opponent_goal)
311 
312         move_reward = np.dot(unit_ball_opponent_goal, ball_velxy)
313 
314         move_reward = np.clip(move_reward / Ball.max_velxy_norm,
               -1.0, 1.0)
315         return move_reward
```

A.2 *Reaplay Buffer* responsável por armazenar os estados e ações

Código A.2 – Código de Python

```
import math
import torch
import random
import numpy as np
from collections import deque, namedtuple
from .utils import angle_between

# Named tuple for storing experience steps gathered in training
Experience = namedtuple(
    'Experience',
    field_names=['state', 'action', 'reward', 'done',
       'next_state'],
)


class AbsoluteState:
    def __init__(self, state_size: int):
        self.state_size = state_size

    def process(self, state):
        robot2control = 0

        robot = state['ally'][robot2control]['pos_xy']
        angle_robot = state['ally'][robot2control]['theta']
        v_robot = state['ally'][robot2control]['vel_xy']
        w_robot = state['ally'][robot2control]['w']
        ball = state['ball']['pos_xy']
        v_ball = state['ball']['vel_xy']

        goal = np.array([0.75,0])

        near_wall1 = 1 - (-robot[0] + 0.75)
        near_wall2 = 1 - (robot[0] + 0.75)
        near_wall3 = 1 - (-robot[1] + 0.65)
        near_wall4 = 1 - (robot[1] + 0.65)

        processed_state = [ robot[0], robot[1],
                            ball[0], ball[1],
                            v_robot[0], v_robot[1],
                            v_ball[0],v_ball[1],
                            goal[0], goal[1],
                            angle_robot, w_robot / 6.28,
                            near_wall1, near_wall2,
                            near_wall3, near_wall4,
                            ]

```

```
46         return processed_state
47 
48 class RelativeState:
49     def __init__(self, state_size: int):
50         self.state_size = state_size
51 
52         self.len_field = np.array([1.5, 1.3])
53         self.diagonal_len_field =
              np.sqrt(np.sum(self.len_field**2))
54 
55     def process(self, state):
56         robot2control = 0
57 
58         robot = state['ally'][robot2control]['pos_xy']
59         angle_robot = state['ally'][robot2control]['theta']
60         v_robot = state['ally'][robot2control]['vel_xy']
61         w_robot = state['ally'][robot2control]['w']
62 
63         robot_vel_unit = np.array([math.cos(angle_robot),
              math.sin(angle_robot)])
64 
65         ball = state['ball']['pos_xy']
66         v_ball = state['ball']['vel_xy']
67 
68         opponent_goal = np.array([0.75,0.0])
69         ally_goal = -opponent_goal
70 
71         robot_ball = ball - robot
72         norm_robot_ball = np.linalg.norm(robot_ball)
73         velrobot_robotballangle = angle_between(robot_vel_unit,
              robot_ball)
74 
75         ball_opponentgoal = opponent_goal - ball
76         ball_allygoal = ally_goal - ball
77 
78         robot_allygoal = ally_goal - robot
79         norm_robot_allygoal = np.linalg.norm(robot_allygoal)
80 
81         vrobot_ballallygoal_angle = angle_between(robot_vel_unit,
              ball_allygoal)
82 
83         near_wall1 = -robot[0] + 0.75
84         near_wall2 = robot[0] + 0.75
85         near_wall3 = -robot[1] + 0.65
86         near_wall4 = robot[1] + 0.65
87 
88         processed_state = [ velrobot_robotballangle,
              norm_robot_ball,
89                             v_robot[0], v_robot[1],
90                             w_robot / 6.28, angle_robot,
91                             near_wall1, near_wall2,
92                             near_wall3, near_wall4,
```

```
93                                norm_robot_allygoal,
94                                vrobot_ballallygoal_angle,
95                                v_ball[0], v_ball[1],
96                                ]
97
98          return processed_state
99
100 class Memory(object):
101     def __init__(self, memory_size: int, is_per: bool =False) ->
            None:
102         super().__init__()
103         self.memory = deque(maxlen=memory_size)
104         self.is_per = is_per
105
106     def append(self, experience: Experience) -> None:
107         self.memory.append(experience)
108
109     def __len__(self):
110         return len(self.memory)
111
112     def sample(self, batch_size):
113
114         state_batch = []
115         action_batch = []
116         reward_batch = []
117         done_batch = []
118         next_state_batch = []
119
120         batch = random.sample(self.memory, batch_size)
121
122         for state, action, reward, done, next_state in batch:
123             state_batch.append(state)
124             action_batch.append(action)
125             reward_batch.append([reward])
126             done_batch.append([done])
127             next_state_batch.append(next_state)
128
129         return (
130             None,
131             torch.from_numpy(np.array(state_batch)).float(),
132             torch.from_numpy(np.array(action_batch)).float(),
133             torch.from_numpy(np.array(reward_batch,
                    dtype=np.float32)).float(),
134             torch.from_numpy(np.array(done_batch,
                    dtype=np.float32)).float(),
135             torch.from_numpy(np.array(next_state_batch)).float(),
136             None,
137         )
```

A.3 Arquiteturas das redes neurais utilizadas na monografia

Código A.3 – Código de Python

```python
import torch
from torch import nn
from omegaconf import DictConfig
import hydra
import numpy as np
from typing import List
from torch.optim.optimizer import Optimizer
from .data import Experience
from .utils import weights_init
import torch.nn as nn

device = torch.device('cuda:1')

class BasicBlock(nn.Module):
    def __init__(self, hidden_size: int, activation = nn.Mish):
        super().__init__()
        self.block = nn.Sequential(
            nn.Linear(hidden_size,hidden_size),
            activation(),
            nn.Linear(hidden_size,hidden_size),
            activation(),
            )

    def forward(self, x):
        return self.block(x) + x



class DDPGActor(nn.Module):
    """
    Args:
        obs_size: observation/state size of the environment
        n_actions: number of discrete actions available in the
           environment = (V,W)
    """
    def __init__(self, obs_size: int, n_actions: int,
       activation=nn.Mish):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(obs_size, 512),
            activation(),
            BasicBlock(hidden_size=512, activation=activation),
            nn.Linear(512, n_actions),
            nn.Tanh(),
            )

```

```
45      def forward(self, x):
46          x = self.layers(x)
47
48          return x
49
50  class DDPGCritic(nn.Module):
51      def __init__(self, obs_size: int,  n_actions: int,
            activation=nn.Mish):
52
53          super().__init__()
54
55          self.layers = nn.Sequential(
56              nn.Linear(obs_size+n_actions, 512),
57              activation(),
58              BasicBlock(512, activation=activation),
59              nn.Linear(512, 1),
60          )
61
62      def forward(self, state, action):
63          x = torch.cat([state, action], 1)
64          x = self.layers(x)
65
66          return x
67
68  class TD3Critic(nn.Module):
69      def __init__(self, obs_size: int,  n_actions: int,
            activation=nn.Mish):
70
71          super().__init__()
72
73          self.layers1 = nn.Sequential(
74              nn.Linear(obs_size+n_actions, 512),
75              activation(),
76              BasicBlock(512, activation=activation),
77              nn.Linear(512, 1),
78          )
79
80          self.layers2 = nn.Sequential(
81              nn.Linear(obs_size+n_actions, 512),
82              activation(),
83              BasicBlock(512, activation=activation),
84              nn.Linear(512, 1),
85          )
86
87      def forward(self, state, action):
88          x1 = torch.cat([state, action], 1)
89          x1 = self.layers1(x1)
90
91          x2 = torch.cat([state, action], 1)
92          x2 = self.layers2(x2)
93
94          return x1, x2
```

```
95
96      def C1(self, state, action):
97          x1 = torch.cat([state, action], 1)
98          x1 = self.layers1(x1)
99
100         return x1
101
102 class DDPGAgent():
103     def __init__(
104         self,
105         model_conf: DictConfig,
106         memory_conf: DictConfig,
107         optimizer_conf: DictConfig,
108         scheduler_conf: DictConfig,
109         process_state_conf: DictConfig,
110         max_grad_norm: DictConfig,
111         gamma: float,
112         tau: float,
113         **kwargs,
114     ):
115
116         self.actor = hydra.utils.instantiate(model_conf.actor)
117         self.actor = self.actor.apply(weights_init)
118         self.actor = self.actor.to(device)
119
120         self.critic = hydra.utils.instantiate(model_conf.critic)
121         self.critic = self.critic.apply(weights_init)
122         self.critic = self.critic.to(device)
123
124         self.process_state =
               hydra.utils.instantiate(process_state_conf)
125         self.memory = hydra.utils.instantiate(memory_conf)
126
127         self.scheduler_conf = scheduler_conf
128         self.optimizer_conf = optimizer_conf
129
130         self.gamma = gamma
131         self.max_grad_norm = max_grad_norm
132
133         self.tau = tau
134
135         self.target_actor =
               hydra.utils.instantiate(model_conf.actor)
136         self.target_actor = self.target_actor.to(device)
137
138         self.target_critic =
               hydra.utils.instantiate(model_conf.critic)
139         self.target_critic = self.target_critic.to(device)
140
141         self.hard_update_target_networks()
142
```

```
143         self.actor_opt, self.critic_opt, self.actor_sch,
               self.critic_sch = self.configure_optimizers()
144 
145     def hard_update_target_networks(self):
146         for target_param, param in
               zip(self.target_actor.parameters(),
               self.actor.parameters()):
147             target_param.data.copy_(param.data)
148         for target_param, param in
               zip(self.target_critic.parameters(),
               self.critic.parameters()):
149             target_param.data.copy_(param.data)
150 
151     def configure_optimizers(self) -> List[Optimizer]:
152         actor_optimizer = hydra.utils.instantiate(
153             self.optimizer_conf.actor,
                   params=self.actor.parameters()
154         )
155 
156         actor_scheduler = hydra.utils.instantiate(
157             self.scheduler_conf.actor, optimizer=actor_optimizer
158         )
159 
160         critic_optimizer = hydra.utils.instantiate(
161             self.optimizer_conf.critic,
                   params=self.critic.parameters()
162         )
163 
164         critic_scheduler = hydra.utils.instantiate(
165             self.scheduler_conf.critic, optimizer=critic_optimizer
166         )
167 
168         return (actor_optimizer, critic_optimizer,
               actor_scheduler, critic_scheduler)
169 
170     def get_action(self, state):
171         state = torch.tensor([self.process_state.process(state)],
               dtype=torch.float).to(device)
172 
173         self.actor.eval()
174 
175         action = self.actor(state)
176         self.actor.train()
177 
178         action = action.detach().cpu().numpy()
179         action = np.clip(action, -1, 1)[0]
180 
181         return action
182 
183     def train_networks(self, batch_size, step, reward):
184 
```

```
185         idxs, states, actions, rewards, dones, next_states, _ =
                self.memory.sample(batch_size)
186 
187         states = states.to(device)
188         actions = actions.to(device)
189         rewards = rewards.to(device)
190         dones = dones.to(device)
191         next_states = next_states.to(device)
192 
193         # critic
194         q_values = self.critic(states, actions)
195         next_actions = self.target_actor(next_states)
196         next_q = self.target_critic(next_states, next_actions)
197 
198         q_targets = rewards + (1-dones)*self.gamma * next_q
199 
200         critic_loss = nn.MSELoss()(q_targets, q_values)
201         self.critic_opt.zero_grad()
202         critic_loss.backward()
203         torch.nn.utils.clip_grad_norm_(self.critic.parameters(),
                self.max_grad_norm.critic)
204         self.critic_opt.step()
205 
206         # actor
207         actor_loss = -self.critic(states,
                self.actor(states)).mean()
208         self.actor_opt.zero_grad()
209         actor_loss.backward()
210         torch.nn.utils.clip_grad_norm_(self.actor.parameters(),
                self.max_grad_norm.actor)
211         self.actor_opt.step()
212 
213         self.soft_update_actor_target_network()
214         self.soft_update_critic_target_network()
215 
216         output = {
217             'critic_loss': critic_loss.cpu().detach().numpy(),
218             'actor_loss': actor_loss.cpu().detach().numpy(),
219             'actor_lr': self.actor_opt.param_groups[0]['lr'],
220             'critic_lr': self.critic_opt.param_groups[0]['lr'],
221             'memory_size': len(self.memory),
222             'reward': reward,
223         }
224 
225         return output
226 
227     def soft_update_actor_target_network(self):
228         for target_param, param in
                zip(self.target_actor.parameters(),
                self.actor.parameters()):
229             target_param.data.copy_(param.data * self.tau +
                    target_param.data * (1.0 - self.tau))
```

```
230
231     def soft_update_critic_target_network(self):
232         for target_param, param in
              zip(self.target_critic.parameters(),
              self.critic.parameters()):
233             target_param.data.copy_(param.data * self.tau +
                    target_param.data * (1.0 - self.tau))
234
235     def save_experience(self, state, action, reward, done,
            new_state):
236
237         exp = Experience(self.process_state.process(state),
238                          action,
239                          reward,
240                          done,
241                          self.process_state.process(new_state),
242                          )
243
244         self.memory.append(exp)
245
246     def step_schedulers(self, actor_loss, critic_loss):
247
248         self.actor_sch.step(actor_loss)
249         self.critic_sch.step(critic_loss)
250
251
252
253 class TD3Agent(DDPGAgent):
254     def __init__(
255         self,
256         model_conf: DictConfig,
257         memory_conf: DictConfig,
258         optimizer_conf: DictConfig,
259         scheduler_conf: DictConfig,
260         process_state_conf: DictConfig,
261         max_grad_norm: DictConfig,
262         gamma: float,
263         tau: float,
264         policy_delay: float,
265         target_noise: DictConfig,
266         noise_clip: float,
267     ):
268
269         super().__init__(model_conf=model_conf,
270                           memory_conf=memory_conf,
271                           optimizer_conf=optimizer_conf,
272                           scheduler_conf=scheduler_conf,
273                           process_state_conf=process_state_conf,
274                           max_grad_norm=max_grad_norm,
275                           gamma=gamma,
276                           tau=tau)
277
```

```
278         self.noise_clip = noise_clip
279         self.target_noise = target_noise
280         self.policy_delay = policy_delay
281         self.actor = hydra.utils.instantiate(model_conf.actor)
282         self.actor = self.actor.to(device)
283 
284         self.critic = hydra.utils.instantiate(model_conf.critic)
285         self.critic = self.critic.to(device)
286 
287         self.target_actor =
               hydra.utils.instantiate(model_conf.actor)
288         self.target_actor = self.target_actor.to(device)
289 
290         self.target_critic =
               hydra.utils.instantiate(model_conf.critic)
291         self.target_critic = self.target_critic.to(device)
292 
293         self.hard_update_target_networks()
294 
295         self.actor_opt, self.critic_opt, self.actor_sch,
               self.critic_sch = self.configure_optimizers()
296 
297     def train_networks(self, batch_size, step, reward):
298 
299         idxs, states, actions, rewards, dones, next_states, _ =
               self.memory.sample(batch_size)
300 
301         states = states.to(device)
302         actions = actions.to(device)
303         rewards = rewards.to(device)
304         dones = dones.to(device)
305         next_states = next_states.to(device)
306 
307         q_values1, q_values2 = self.critic(states, actions)
308 
309         with torch.no_grad():
310             noise = (
311                     torch.randn_like(actions) * self.target_noise
312                 ).clamp(-self.noise_clip, self.noise_clip)
313 
314             next_actions = self.target_actor(next_states) + noise
315             next_actions = torch.clamp(next_actions, -1.0, 1.0)
316 
317             next_q1, next_q2 = self.target_critic(next_states,
                   next_actions)
318             next_q = torch.min(next_q1, next_q2)
319 
320             q_targets = rewards + (1.0 - dones) * self.gamma *
                   next_q
321 
322         critic_loss = nn.MSELoss()(q_targets, q_values1) +
               nn.MSELoss()(q_targets, q_values2)
```

```
323         self.critic_opt.zero_grad()
324         critic_loss.backward()
325         torch.nn.utils.clip_grad_norm_(self.critic.parameters(),
                self.max_grad_norm.critic)
326         self.critic_opt.step()
327 
328         # actor
329         actor_loss = -self.critic.C1(states,
                self.actor(states)).mean()
330 
331         if step % self.policy_delay == 0:
332             self.soft_update_actor_target_network()
333             self.soft_update_critic_target_network()
334 
335             self.actor_opt.zero_grad()
336             actor_loss.backward()
337             torch.nn.utils.clip_grad_norm_(self.actor.parameters(),
                    self.max_grad_norm.actor)
338             self.actor_opt.step()
339 
340         if self.memory.is_per:
341             errors = np.abs((q_values1 -
                    q_targets).detach().cpu().numpy())
342             for idx, error in zip(idxs, errors):
343                 self.memory.update(idx, error)
344 
345         output = {
346             'critic_loss': critic_loss.cpu().detach().numpy(),
347             'actor_loss': actor_loss.cpu().detach().numpy(),
348             'actor_lr': self.actor_opt.param_groups[0]['lr'],
349             'critic_lr': self.critic_opt.param_groups[0]['lr'],
350             'memory_size': len(self.memory),
351             'reward': reward,
352         }
353 
354         return output
```

A.4 Arquivo principal responsável por dar início ao treinamento

Código A.4 – Código de Python

```
1 import hydra
2 import numpy as np
3 import shutil
4 import os
5 
6 os.environ['CUDA_LAUNCH_BLOCKING'] = "1"
7 
```

```
from src.utils import save_onnx
from numpy.core.fromnumeric import mean
import pytorch_lightning as pl
from src.noise import OUNoise
from collections import defaultdict
import torch
from pathlib import Path
from torch.utils.tensorboard import SummaryWriter

writer = SummaryWriter()

def save_model(agent, epoch, loss, dest_dir):
    actor = (agent.actor, str(Path(dest_dir) /
       f"actor_{epoch}.pt"))
    critic = (agent.critic, str(Path(dest_dir) /
       f"critic_{epoch}.pt"))

    for net, path in (actor, critic):
        torch.save({
                'epoch': epoch,
                'model_state_dict': net.state_dict(),
                'loss': loss,
                }, path)

def load_model(agent, src_dir, epoch="4150"):
    device = torch.device('cuda:1')

    actor_path = str(Path(src_dir) / f"actor_{epoch}.pt")
    state_dict = torch.load(actor_path)["model_state_dict"]
    agent.actor.load_state_dict(state_dict)
    agent.actor = agent.actor.to(device)

    critic_path = str(Path(src_dir) / f"critic_{epoch}.pt")
    state_dict = torch.load(critic_path)["model_state_dict"]
    agent.critic.load_state_dict(state_dict)
    agent.critic = agent.critic.to(device)

    agent.hard_update_target_networks()

def report2tensorboard(training_step_outputs, idx):
    buffer = defaultdict(list)

    for out in training_step_outputs:
        for key, val in out.items():
            buffer[key].append(val)

    avg_buffer = dict()

    for key, list_val in buffer.items():
        avg_value = mean(list_val)
        avg_buffer[key] = avg_value

```

```
58            writer.add_scalar(key, avg_value, idx)
59
60        return avg_buffer
61
62    @hydra.main(version_base='1.2', config_path='config',
          config_name='config')
63    def main(cfg):
64        pl.seed_everything(cfg.seed)
65        if not cfg.only_get_onnx:
66            logger = pl.loggers.TensorBoardLogger(cfg.logdir)
67
68        env = hydra.utils.instantiate(cfg.env)
69
70        act_noise = cfg.act_noise
71
72        agent = hydra.utils.instantiate(
73            cfg.agent,
74            model_conf=cfg.model,
75            memory_conf=cfg.memory,
76            optimizer_conf=cfg.optimizer,
77            scheduler_conf=cfg.scheduler,
78            process_state_conf=cfg.process_state,
79            target_noise=cfg.target_noise,
80            noise_clip=cfg.noise_clip,
81            _recursive_=False,
82            )
83
84        load_model(agent, src_dir="outputs/2023-06-14/10-19-30",
              epoch=1500)
85        # load_model(agent, src_dir="outputs/2023-06-22/08-40-06",
              epoch=2000)
86        # load_model(agent, src_dir="outputs/2023-06-24/16-01-25",
              epoch=1000)
87
88        try:
89            if not cfg.only_get_onnx:
90
91                noise = OUNoise(cfg.action_size)
92                for epoch in range(cfg.max_epochs):
93                    training_step_outputs = []
94                    done = False
95
96                    episode_reward = 0
97
98                    state = env.reset_random_init_pos()
99                    step = 0
100
101                   while not done:
102                       action = agent.get_action(state)
103                       env_action = np.reshape(action, (-1,2))
104
```

```
105                     noise = np.random.normal(0,1,
                           size=env_action[0].shape)*act_noise
106                     env_action[0] = noise + env_action[0]
107                     env_action[0] = np.clip(env_action[0], -1,1)
108                     env_action[0] =
                           env_action[0]*np.array([env.max_v,
                           env.max_w])
109
110                     new_state, reward, done = env.step(env_action)
111
112                     agent.save_experience(state, action, reward,
                           done, new_state)
113
114                     if len(agent.memory) > cfg.batch_size:
115                         out = agent.train_networks(cfg.batch_size,
                               step, reward)
116                         training_step_outputs.append(out)
117
118                     state = new_state
119                     step += 1
120                     episode_reward += reward
121
122                 avg = report2tensorboard(training_step_outputs,
                       epoch)
123                 agent.step_schedulers(
124                         actor_loss=avg["actor_loss"],
125                         critic_loss=avg["critic_loss"],
126                         )
127
128                 writer.add_scalar("reward", episode_reward, epoch)
129                 writer.add_scalar("epoch", epoch, epoch)
130                 writer.add_scalar("max_dist", env.max_dist, epoch)
131
132                 if epoch and not epoch % cfg.sync_save_onnx and
                       not cfg.is_debug:
133                     save_onnx(agent.actor, logger.save_dir,
                           cfg.process_state.state_size)
134
135                 if env.max_dist < 1.50:
136                     env.max_dist += 0.10
137
138                 if epoch and not epoch % cfg.sync_save_pt:
139                     avg_sum = avg["actor_loss"]+avg["critic_loss"]
140                     save_model(agent=agent,
141                             epoch=epoch,
142                             loss=avg_sum,
143                             dest_dir=logger.save_dir,
144                             )
145
146     except KeyboardInterrupt:
147         print('KeyboardInterrupt raised.')
148     finally:
```

```
149         if not cfg.is_debug:
150             if cfg.only_get_onnx:
151                 save_onnx(agent.actor, "onnx",
                        cfg.process_state.state_size)
152             else:
153                 save_onnx(agent.actor, logger.save_dir,
                        cfg.process_state.state_size)
154 
155         writer.flush()
156         writer.close()
157 
158 if __name__ == '__main__':
159     main()
```

A.5 Arquivo responsável pela etapa de validação dos agentes

Código A.5 – Código de Python

```
1  import time
2  from firasim_client.env import MarkovDecisionProcess
3  
4  if __name__ == "__main__":
5      loop_time = 0.008
6      env = MarkovDecisionProcess()
7      state = env.reset_random_init_pos()
8  
9      last = time.time()
10     last_packet = None
11 
12     while True:
13         if (time.time() - last) < loop_time:
14             packet = env.vision.read()
15             if packet is not None:
16                 last_packet = packet
17         else:
18             break
19 
20     while True:
21         time.sleep(0.1)
22         next_state, reward, done = env.step([[0.0, 0.0]])
23         # print(next_state)
24         # print(reward)
25         # print(done)
```

A.6 Funções secundárias necessárias para a implantação de utilidades ao longo do projeto

Código A.6 – Código de Python

```
import json
import torch
import torch.onnx
import numpy as np
from torch import nn
from pathlib import Path


def load_json(path):
    with open(path, "r") as f:
        return json.load(f)


def save_json(data, path):
    with open(path, "w") as f:
        return json.dump(data, f, indent=4)

def save_onnx(model, save_path, state_size: int):
    try:
        conv2onnx(model, Path(save_path) / 'last-model.onnx',
           size=(state_size, 1))
    except IndexError:
        print('Best model not found, skipping torch save')


def conv2onnx(model, outpath, size):

    w,h = size
    model.eval()
    device = torch.device('cuda:1')
    x = torch.randn((h, w)).to(device)

    torch.onnx.export(
        model,  # model being run
        x,  # model input (or a tuple for multiple inputs)
        outpath,  # where to save the model (can be a file or
           file-like object)
        export_params=True,  # store the trained parameter weights
           inside the model file
        opset_version=11,  # the ONNX version to export the model
           to
        do_constant_folding=True,  # whether to execute constant
           folding for optimization
        input_names=["input"],  # the model's input names
        output_names=["output"],  # the model's output names
        dynamic_axes={
```

```
42                "input": {0: "batch_size"},  # variable lenght axes
43                "output": {0: "batch_size"},
44            },
45        )
46
47
48  def angle_between(v1, v2):
49      """ Returns the angle in radians between vectors 'v1' and 'v2'
           in rads.
50      The angle can be positive or negative
51      """
52      def unit_vector(vector):
53          """ Returns the unit vector of the vector."""
54          return vector / np.linalg.norm(vector)
55
56      v1_u = unit_vector(v1)
57      v2_u = unit_vector(v2)
58      angle_v1, angle_v2 = np.arctan2([v1_u[1],v2_u[1]], [v1_u[0],
           v2_u[0]])
59
60      delta = angle_v1 - angle_v2
61
62      if delta > np.pi:
63          delta = delta - 2*np.pi
64      elif delta < -np.pi:
65          delta = 2*np.pi + delta
66
67      return -delta
68
69  def normalized_angle(v1, v2):
70      # Receive 2 vectors (the order is important) and return the
           the range (-1,1)
71      return angle_between(v1,v2)/np.pi
72
73
74  def norm_grad(model):
75      total_norm = 0
76      parameters = [p for p in model.parameters() if p.grad is not
           None and p.requires_grad]
77      for p in parameters:
78          param_norm = p.grad.detach().data.norm(2)
79          total_norm += param_norm.item() ** 2
80      total_norm = total_norm ** 0.5
81      return total_norm
82
83  def weights_init(m):
84      # for every Linear layer in a model..
85      if isinstance(m, nn.Conv2d) or isinstance(m, nn.Linear):
86          m.weight.data.normal_(0.0, 0.02)
87          m.bias.data.fill_(0)
```