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

Matheus dos Santos Lima

APLICAÇÃO DO ALGORITMO DQN A UM ROBÔ DELTA

Santa Maria, RS
2022

Matheus dos Santos Lima

APLICAÇÃO DO ALGORITMO DQN A UM ROBÔ DELTA

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

Orientador: Prof. Dr. Daniel Fernando Tello Gamarra

Santa Maria, RS
2022

Matheus dos Santos Lima

APLICAÇÃO DO ALGORITMO DQN A UM ROBÔ DELTA

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

Aprovado em 18 de Novembro de 2022:

_____

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

_____

Leonardo Ramos Emmendorfer, Dr. (UFSM)

_____

Anselmo Rafael Cukla, Dr. (UFSM)

Santa Maria, RS
2022

RESUMO

APLICAÇÃO DO ALGORITMO DQN A UM ROBÔ DELTA

AUTOR: Matheus dos Santos Lima
ORIENTADOR: Daniel Fernando Tello Gamarra

O presente trabalho tem como objetivos a modelagem da cinemática de um robô paralelo Delta e a aplicação do algoritmo DQN em um ambiente desenvolvido com base neste modelo, o agente treinado em tal ambiente aprende por meio de tentativa e erro recebendo recompensas positivas quando acerta e negativas quando erra. Assim, por meio do processo iterativo o agente desenvolve novas políticas, isto é, novas regras sobre como tomar ações dado um novo estado deste ambiente. Um ambiente de simulação cujo objetivo é posicionar corretamente o efetuador do robô Delta em um dado ponto no espaço é proposto, este ambiente consiste em um algoritmo que utiliza o modelo matemático do mecanismo para gerar renderizações do robô toda vez que o agente executa uma nova ação. Foi possível obter 80% de acertos durante o treinamento o que pode ser observado no gráfico de recompensas no capítulo de resultados. Com a implementação do algoritmo no ambiente do robô Delta observou-se que a solução implementada é capaz de resolver a grande maioria dos episódios aos quais o agente é apresentado, no entanto, em alguns pontos específicos, o agente falha em obter uma solução adequada, isso ocorre geralmente em pontos de singularidade, isto é, regiões nas quais o mecanismo pode perder sua rigidez estrutural ou não conseguir obter uma solução única.

ABSTRACT

DEEP REINFORCEMENT LEARNING APPLIED TO A DELTA ROBOT

AUTHOR: Matheus dos Santos Lima
ADVISOR: Daniel Fernando Tello Gamarra

The present work aims to model the kinematics of a Delta parallel robot and the application of the DQN algorithm in an environment developed based on this model, the agent trained in such environment learns through trial and error, receiving positive rewards when it correctly guesses the next step and negative rewards when it misses. Thus, through the iterative process, the agent develops new policies, that is, new rules on how to take actions given a new state of the environment. A simulation environment whose objective is to correctly position the Delta robot's effector at a given point in space is proposed, this environment consists of an algorithm that uses the mechanism mathematical model to generate renders of the robot every time the agent takes an action. It was possible to obtain 80% of correct answers during training, which can be observed in the rewards graph in the results chapter. With the implementation of the algorithm in the Delta robot environment, it was possible to observe that the implemented solution can solve most of the episodes to which the agent is presented, however, in some specific points, the agent fails to obtain a solution. This usually occurs at singularity points, that is, regions where the mechanism may lose its structural rigidity or fail to obtain a unique solution.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

AI	Inteligência Artificial
DL	Aprendizado Profundo
DP	Programação Dinâmica
DQN	Deep Q Networks
DRL	Aprendizado por Reforço Profundo
MDP	Processos De Decisão De Markov
ML	Aprendizado De Máquina
RL	Aprendizado Por Reforço
TD	Diferença-Temporal

SUMÁRIO

1 INTRODUÇÃO

Aprendizado por reforço significa aprender a mapear estados que podem ser: posições, velocidades, acelerações, distâncias, entre outros, para ações que podem ser: tomadas de decisão baseadas nos estados anteriores, a fim de alcançar um objetivo, ou em outras palavras, maximizar um sinal de recompensa determinado por um ambiente de aprendizado. O agente não sabe que ações tomar para obter melhores resultados, assim, este deve aprender uma política de ações através da interação repetida com o ambiente e com isso, determinar quais ações retornam as maiores recompensas, através de tentativa e erro (SUTTON; BARTO, 2018).

Importantes resultados do aprendizado de máquina (*Machine Learning* - ML) se deram através da implementação de redes neurais para solução de problemas de aprendizado por reforço. Como exemplo, pode-se citar a implementação de algoritmos de Aprendizado por Reforço Profundo (do inglês *Deep Reinforcement Learning* - DRL) para os ambientes do videogame Atari 2600® utilizando o algoritmo de aprendizado por reforço Q com redes neurais desenvolvido por Mnih e colaboradores (2015).

Um outro exemplo de aplicação de DRL é o projeto AlphaGo® da empresa Deepmind® que foi capaz de vencer um campeão mundial no jogo de tabuleiro Go® (SILVER, 2016). Além desse, o projeto AlphaStar® da mesma empresa foi capaz de vencer no jogo de computador Starcraft II® em nível profissional, tudo isso foi possível devido aos avanços em aprendizado de máquina e Deep Learning (ALPHASTAR, 2019).

Nesse contexto, este trabalho visa contribuir para o crescente desenvolvimento da área de aprendizado por reforço. Mais especificamente, a implementação de um algoritmo de aprendizado por reforço em um mecanismo robótico de natureza paralela, um robô paralelo Delta (BONEV, 2001). Com base nos métodos de aprendizado por reforço que utilizam funções de otimização Q com redes neurais profundas, o método Deep Q Networks (DQN) foi implementado em um ambiente desenvolvido para o robô Delta.

1.1 OBJETIVO GERAL

Treinar um agente implementando o algoritmo DQN em um ambiente constituído pelo mecanismo do robô Delta cuja tarefa é posicionar o efetuador deste em um determinado ponto gerado aleatoriamente no espaço.

1.2 OBJETIVOS ESPECÍFICOS

- Derivar as equações de cinemática Direta e Inversa do robô Delta;
- Provar o equacionamento obtendo resultados de geração de espaço de trabalho do robô Delta e trajetórias;
- Desenvolver um ambiente para a tarefa de posicionamento do robô Delta utilizando as equações de cinemática;
- Implementar um algoritmo DQN no ambiente de simulação;
- Obter resultados de treinamento do robô no ambiente de simulação;
- Demonstrar visualmente o comportamento do robô no ambiente de simulação.

1.3 JUSTIFICATIVA

Em anos recentes tem-se observado uma grande variedade de aplicações de aprendizado por reforço, se pode destacar aplicações em jogos eletrônicos, solução de problemas clássicos da teoria de controle, além de diversas aplicações em robôs móveis, dentre outros.

O presente trabalho é motivado pela crescente evolução da área de aprendizado por reforço profundo e suas diversas aplicações na grande área da robótica. Mais especificamente tem-se o desejo de contribuir com o desenvolvimento da área com a aplicação do algoritmo DQN na solução da cinemática inversa de um robô paralelo, o robô Delta.

1.4 SÍNTESE DOS PROXIMOS CAPÍTULOS

Esta subseção descreve brevemente o conteúdo de cada capítulo do texto, dessa forma, o leitor saberá de antemão o que esperar de cada seção. A fundamentação teórica do trabalho é descrita no capítulo 2, este descreve brevemente o aprendizado de máquina a partir do aprendizado supervisionado, aprendizado não-supervisionado e aprendizado por reforço. Uma introdução ao aprendizado por reforço é apresentada, bem como a base teórica do aprendizado por reforço profundo, além de uma explicação sobre o algoritmo DQN, o tema do presente trabalho, uma discussão acerca dos robôs paralelos finaliza este capítulo. As ferramentas e metodologia de projeto são descritas no capítulo 3, uma descrição do ambiente de desenvolvimento foi inserida além de informações pertinentes à linguagem de programação utilizada bem como as bibliotecas que foram utilizadas para que o trabalho fosse implementado. Além disso, este descreve a arquitetura de rede utilizada bem como as especificações tais como número de camadas, função de ativação e, também um fluxograma do algoritmo. Os resultados são apresentados no capítulo 4, foram descritos os resultados de treinamento, testes com pontos

aleatórios e testes com pontos fixos. Através dos gráficos de recompensa apresentados e a tabela de resultados dos testes em pontos fixos, é possível se observar os resultados obtidos. O anexo A apresenta informações acerca do acesso ao código via GitHub, além de conter link do vídeo do robô no ambiente, o anexo B apresenta o código para treinamento do robô, os anexos C e D, são utilizados para testes no ambiente com pontos aleatórios e pontos fixos, respectivamente.

2 REFERENCIAL TEÓRICO

Um algoritmo de aprendizado de máquina é um método programático capaz de aprender a partir de dados (GOODFELLOW; BENGIO; COURVILLE, 2016). De acordo com MITCHELL (1997): “Um programa de computador é dito ser capaz de aprender a partir de experiências *E* com respeito a uma dada tarefa *T* e medido por uma performance *P*, se sua performance na tarefa *T*, tal como medido por *P*, melhora com a experiência *E*.”

Pode-se imaginar uma grande gama de tarefas para essa definição, e de fato ela é capaz de encapsular a maior parte das aplicações de aprendizado de máquina. É possível dividir o aprendizado de máquina em três grandes categorias: aprendizado supervisionado, aprendizado não supervisionado e aprendizado por reforço.

2.1 APRENDIZADO POR REFORÇO

Em termos gerais, o aprendizado por reforço é uma forma de aprender por tentativa e erro. Esse tipo de problema não possui um conjunto de dados rotulados ou mesmo um conjunto de dados. O objetivo dessa metodologia é agir em um ambiente de aprendizagem e de maneira iterativa encontrar uma solução para o problema (GÉRON, 2017).

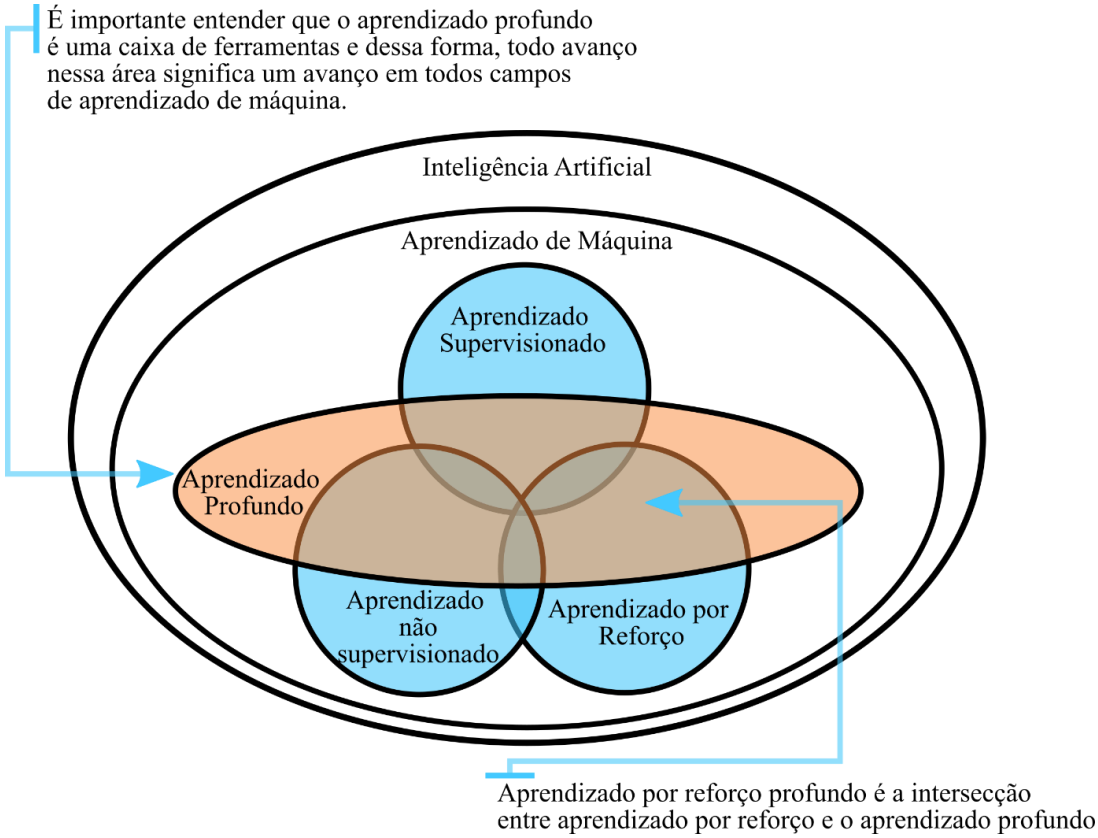
De acordo com Sutton e Barto (2018), o RL se baseia em determinar o que é preciso fazer para maximizar um sinal numérico de recompensa, isto é, como mapear situações para ações, a fim de se obter a maior recompensa em uma dada condição. O **agente** (que interage e aprende) não sabe que ações deve tomar, deve descobrir por conta própria através de tentativa e erro, testando diferentes ações para obter a maior **recompensa** do **ambiente** (que muda toda vez que o agente executa uma ação). Nos problemas mais desafiadores, tem-se que as **ações** não só afetam a recompensa imediata, como também o próximo estado e consequentemente todas as recompensas posteriores. Estas duas características de tentativa e erro e recompensa tardia são dois dos mais importantes aspectos distintos do RL.

No contexto de RL, a inteligência deve ser entendida como: o aprendizado que o agente obtém após um número de tentativas e erros dentro de um ambiente, este determina uma maneira de gratificá-lo ou puni-lo de acordo com suas **ações** com uma **recompensa** numérica. De maneira formal, o problema de aprendizado por reforço se baseia na teoria de sistemas dinâmicos, mais especificamente, o controle ótimo de um processo de decisões de Markov finito que não é totalmente conhecido (SUTTON; BARTO, 2018).

2.2 APRENDIZADO POR REFORÇO PROFUNDO

O aprendizado profundo (*Deep Learning* - DL) não é um tema separado de aprendizado de máquina, mas sim um conjunto de técnicas e métodos de rede neurais para a solução de problemas de aprendizado de máquina, seja aprendizado supervisionado, não-supervisionado ou por reforço (MORALES, 2020). A Figura 1 mostra como são divididos os campos de aprendizado de máquina e aprendizado profundo.

Figura 1 - Divisão dos campos de aprendizado de máquina e aprendizado profundo



Fonte: Adaptado de (MORALES, 2020).

Quando aplicado em problemas de aprendizado por reforço (*Reinforcement Learning* - RL) temos o aprendizado por reforço profundo que nada mais é que a utilização de técnicas de DL em tarefas de RL. Assim o campo de RL como observado na Figura 1 envolve o campo de DRL.

Tal como abordado anteriormente, importantes avanços no campo de ML se deram no campo de DRL, tanto avanços em pesquisa como aplicações práticas, desde a robótica passando por complicados sistemas de operações no mercado financeiro até aplicações militares (LOCKHEEDMARTIN, 2020).

O DRL trata da implementação de redes neurais artificiais em problemas de RL, tal implementação envolve o desenvolvimento de programas de computador capazes de solucionar problemas que requerem a iteração de um **agente** com um **ambiente**. No contexto de RL, a inteligência deve ser entendida como: o aprendizado que o agente obtém após um número de tentativas e erros dentro de um ambiente, este determina uma maneira de gratificá-lo ou puni-lo de acordo com suas **ações** com uma **recompensa** numérica. De maneira formal, o problema de aprendizado por reforço se baseia na teoria de sistemas dinâmicos, mais especificamente, o controle ótimo de um processo de decisões de Markov finito que não é totalmente conhecido (SUTTON; BARTO, 2018).

No contexto do trabalho atual, o algoritmo implementado o DQN, se utiliza do *framework* de solução de problemas do aprendizado por reforço os processos de decisão de Markov e, também faz uso das redes neurais profundas para desenvolver políticas adequadas ao treinamento executado.

2.3 ALGORITMOS DE APRENDIZADO POR REFORÇO

Dentro de RL existe uma grande variedade de problemas e algoritmos que tem o objetivo resolvê-los. Nesse contexto, serão abordadas as metodologias que tangem o escopo do presente trabalho. Serão descritos os algoritmos que formam a base do DQN. A subseção a seguir trata dos processos de decisão de Markov, que são elemento fundamental no desenvolvimento de algoritmos de Aprendizado por Reforço.

2.3.1 Processos de Decisão Finitos de Markov

Definição de terminologia e propriedades de sistemas de Markov, segundo Sutton e Barto (2018):

- **Agente**: Um agente de aprendizado por reforço é a entidade que é “treinada” para aprender a tomar decisões corretas em um dado ambiente.
- **Ambiente**: O ambiente é tudo o que se encontra fora dos limites do agente, isto é, tudo com o que o agente interage, o agente não manipula o ambiente, este, somente controla suas próprias ações (no caso de um robô, além das características externas, tem-se as características internas deste, ou seja, para o agente, o atuador do robô faz parte do ambiente).
- **Estado**: O estado define a atual situação em que o agente se encontra (pode ser uma posição no espaço ou velocidade das juntas de um robô).

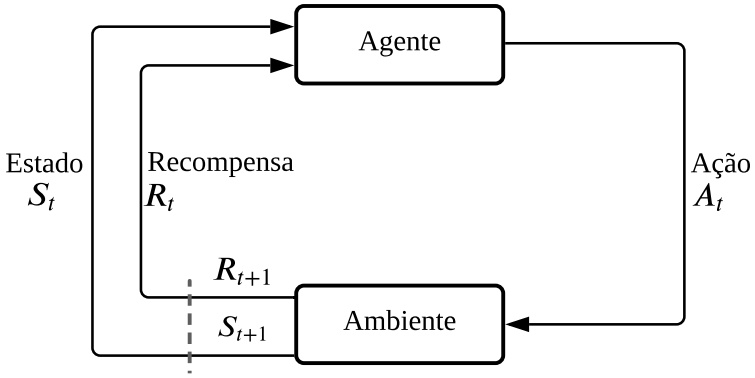
- **Ação**: Trata-se da escolha que o agente faz a cada passo (por exemplo, rotacionar uma junta em 0.5 radianos).
- **Política**: A política se trata do processo analítico desenvolvido pelo agente afim de se determinar uma ação para um estado no qual se encontra, na prática é uma distribuição probabilística de um grupo de ações, quanto melhor a recompensa de uma ação, maior a probabilidade de esta ocorrer novamente.

De acordo com a propriedade de *Markov* o estado atual de um sistema qualquer depende somente do seu estado anterior (passo anterior). E consequentemente, o próximo estado depende somente do estado atual.

Os processos de decisão de Markov (do inglês *Markov Decision Processes* - MDPs) são uma formalização clássica de tomada de decisões sequenciais nos quais as ações influenciam não somente a recompensa imediata, bem como os estados subsequentes, e dessa forma, afetam as recompensas futuras. Assim, as MDPs envolvem uma troca entre a recompensa imediata e a recompensa futura. Nas MDPs estamos interessados em estimar o valor $q*(s,a)$ de cada ação a em cada estado s ou estima-se o valor $v*(s)$ de cada estado dado uma seleção ótima de ações.

As MDPs existem como um *framework* direto do problema de aprendizado por iterações como forma de se chegar a um objetivo. O que toma as decisões é o agente, enquanto tudo aquilo com que este interage e está fora de seu contexto se denomina ambiente. Dessa forma, estes interagem continuamente, o agente tomando ações e o ambiente respondendo apresentando novos estados. O ambiente é também responsável por passar as recompensas ao agente, que com o decorrer do tempo visa maximizar este sinal. A Figura 2 mostra o ciclo de ações e recompensas entre o agente e o ambiente.

Figura 2 - Ciclo de iterações Agente-Ambiente



Fonte: Adaptado de Sutton e Barto, 2018.

A Figura 5 apresenta o ciclo de iterações do agente com o ambiente, onde as variáveis A_t, S_t, R_t representam respectivamente a ação escolhida pelo agente, o estado atual do ambiente e a recompensa que o agente recebeu por tomar a ação, enquanto as variáveis S_{t+1}, R_{t+1}, são respectivamente o estado e a recompensa futura do ambiente.

Dessa forma, ao decorrer do tempo o agente passa a tomar decisões cada vez melhores até que em determinado momento este consegue resolver o ambiente, ou seja, em um determinado conjunto de iterações que são denominados de **episódios** o agente é capaz de tomar decisões corretas que o levam ao objetivo do ambiente em que se encontra (SUTTON; BARTO, 2018).

2.3.2 Programação Dinâmica

O termo Programação Dinâmica (DP – *Dynamic Programming*) se refere a uma coleção de algoritmos que são capazes de computar políticas ótimas dado que se tem um modelo perfeito do ambiente como um MDP. Algoritmos clássicos da DP possuem um limite de uso em Aprendizado por Reforço tanto por necessitar de um modelo perfeito do ambiente como por causa de seu alto custo computacional, apesar disso são teoricamente importantes. DP prove uma fundação essencial para o entendimento dos métodos aqui apresentados, de fato os métodos demonstrados aqui são um esforço de se obter os efeitos de DP, porém, com menos custo computacional e sem a necessidade de um modelo completo do ambiente.

Normalmente assume-se o ambiente como uma MDP finita, ou seja, assume-se que os conjuntos de estados, ações e recompensas são finitos e que suas dinâmicas são determinadas por um conjunto de probabilidades $p(s', r|s, a)$. Apesar de as ideias de DP serem aplicáveis em problemas com espaços de estado e ações contínuos, soluções exatas só são possíveis em casos especiais. A ideia chave de DP, e aprendizado por reforço em geral, é a utilização de funções de valor para organizar e estruturar a busca por boas políticas (SUTTON; BARTO, 2018).

2.3.3 Aprendizado Diferença-Temporal (TD)

Uma das ideias centrais e mais inovadoras no campo de aprendizado por reforço vem dos algoritmos de diferença-temporal (do inglês *Temporal Difference* - TD). O aprendizado TD é uma combinação dos métodos de aprendizado Monte Carlo e a DP. Tal como nos métodos de Monte Carlo, os métodos TD podem aprender diretamente de experiência pura sem a necessidade de um modelo do ambiente. E, da mesma forma que nos métodos de programação dinâmica os métodos TD atualizam as suas estimativas baseadas em parte nas estimativas de

outros aprendizados, sem a necessidade de esperar por um resultado. A relação entre os métodos TD, DP, e Monte Carlo é um tema recorrente na literatura de aprendizado por reforço.

Tanto o método TD quanto Monte Carlo utilizam experiência para resolver o problema de predição. Dada alguma experiência seguida de uma política π, onde o termo política deve ser interpretado como sendo a forma com que o agente está a decidir suas ações conforme os estados que lhe são apresentados, ambos métodos atualizam suas estimativas Vde v_π para os estados não terminais S_t que ocorreram naquela experiência. De maneira geral os métodos de Monte Carlo, esperam até que o retorno da experiência é conhecido, em seguida utilizam o retorno para estimar $V(S_t)$.

Uma das principais vantagens dos métodos TD é o fato de que estes atualizam suas estimativas com base em outras estimativas. Outro fato importante dos métodos TD é que não requerem um modelo do ambiente diferentemente de DP, sua recompensa e estados são distribuições probabilísticas. Além disso, em comparação aos métodos Monte Carlo, TD possuem uma implementação naturalmente incremental e *online*, isto é, atualizam sua política a cada *timestep*, enquanto os métodos de Monte Carlo atualizam sua política a cada fim de episódio (SUTTON; BARTO, 2018).

Estes conceitos formam a base para o algoritmo DQN, foi através deste conhecimento que o algoritmo DQN surgiu e se tornou rapidamente objeto de estudo de muitos pesquisadores. O algoritmo DQN é uma forma mais complexa destes algoritmos mais básicos, este funciona com base na teoria apresentada até aqui.

2.3.4 Algoritmo Q-*Learning*: Controle TD *off-policy*

Uma das grandes descobertas dentro de RL foi o desenvolvimento de um algoritmo TD *off-policy*, o Q-*Learning*, definido da seguinte forma:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \quad (1)$$

Nesse caso, a função de ação-valor Q aproxima diretamente q_*, que é a função de ação-valor ótima, independentemente de qual política está sendo seguida atualmente. Dessa maneira, a análise do algoritmo fica mais simples, o que possibilita provas de convergência logo no começo. A política ainda possui um efeito, já que esta determina quais pares estado-ação serão visitados e atualizados. Apesar disso, tudo que é necessário para convergência é que os pares continuem a ser atualizados, todo método que visa garantir comportamento ótimo deve seguir

este requerimento. Devido a essa suposição, foi demonstrado que Q converge com probabilidade 1 para q_*. O algoritmo Q-*Learning* é apresentado na Tabela 1.

Tabela 1 – Pseudocódigo do algoritmo Q-Learning

Q-*learning* (controle TD *off-policy*) para estimar $\pi \approx \pi_*$
Parâmetros do algoritmo: tamanho do *step* $\alpha \in (0,1]$, pequeno $\varepsilon > 0$
Inicializar $Q(s,a)$, para todo $s \in S^+$, $a \in A(s)$, arbitrariamente a menos que $Q(terminal,.) = 0$
Iterar para cada episódio:
Inicializar S
Iterar para cada *step* do episódio:
Escolher A de S utilizando a política derivada de $Q(e.g, \varepsilon - greedy)$
Tomar ação A, observar R, S'
$Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$
$S \leftarrow S'$
Até que $S\ seja\ terminal$.

Fonte: Adaptado de (SUTTON; BARTO, 2018).

2.4 ALGORITMO DQN

O algoritmo DQN foi implementado no presente trabalho e foi utilizado para resolver o ambiente do robô Delta. Mnih e colaboradores (2015) desenvolveram um agente de RL determinado *Deep Q-network* combinando *Q-Learning* com uma rede neural de convolução que são redes especializadas no processamento de vetores espaciais de dados, como no caso da pesquisa imagens de um emulador. Quando Mnih trabalhou com redes neurais, estas já haviam sido aplicadas em diversos outros problemas com bons resultados, no entanto, foram pouco utilizadas em RL.

Em seu trabalho, Mnih e colaboradores demonstram como um agente DQN foi capaz de alcançar alta performance em uma coleção de diferentes problemas sem a necessidade de dados específicos deles, isto é, sem um modelo do problema em questão. Com isso, demonstraram que ao permitir que agentes interagissem com 49 jogos diferentes de um emulador de Atari 2600, os desenvolveram políticas diferentes para cada jogo, recebendo como entrada imagens do emulador, os agentes foram capazes de jogar em nível igual ou superior ao jogador humano em uma grande quantidade de jogos.

A tarefa a ser considerada no trabalho atual é a de posicionar o efetuador de um robô Delta em um dado ponto no espaço que esteja contido em seu espaço de trabalho. A interação do

agente com o ambiente vai determinar novas políticas de acordo com a ação executada, recompensas recebidas e a política atual dele. O agente é tão somente o algoritmo que recebe estados como entradas e executa uma ação baseada em uma política, isto é, tudo que é parte do mecanismo do robô é consequentemente parte do ambiente. Uma política determina a maneira que o agente escolhe uma ação dado um estado do ambiente, dessa forma, o objetivo principal é obter uma política de ações ótima para todo e qualquer estado que o ambiente apresenta.

Dado que temos um ambiente ε e a cada *timestep* t o agente seleciona uma ação a_t do conjunto de ações possíveis no ambiente, $A = \{1, 2, 3, \dots 15\}$, neste caso temos quinze ações possíveis. A ação escolhida é passada ao simulador, que modifica o estado atual do ambiente e retorna um valor de recompensa r_t, onde a recompensa pode ser positiva ou negativa, positiva se o efetuador diminuiu a distância d até o alvo, negativa se a distância aumentou.

Um estado no ambiente é constituído por $s_t = \{x_g, y_g, z_{g,} x_{ee}, y_{ee}, z_{ee}, \theta_1, \theta_2, \theta_3, d, 0{,}0\}$, onde o sufixo g indica as coordenadas do alvo (*goal*), o sufixo *ee* indica as coordenadas do efetuador do robô (*end-effector*), a variável θ indica os ângulos dos atuadores do robô, estes são medidos em radianos no algoritmo, porém em alguns resultados são apresentados em graus para facilitar a compreensão, por último a variável d indica a distância do efetuador até o alvo, os zeros que aparecem ao final do conjunto de estados servem somente para adequar o perfil do vetor de entrada com o da primeira camada profunda da rede, estes não influenciam em nada no resultado final.

Assim, o objetivo do agente é interagir com o ambiente, selecionando ações de tal maneira a maximizar as recompensas futuras. Se assume como padrão que, as recompensas futuras são descontadas por um fator γ a cada *timestep*, e define-se o retorno futuro descontado no tempo t como $R_t = \Sigma_{t'=t}^{T} \gamma^{t'-t} r_{t'}$, onde T é o *timestep* terminal do ambiente. Define-se a função de ação-valor ótima como $Q^*(s, a)$ como o retorno máximo esperado por seguir uma determinada estratégia, após observadas algumas sequências de estados s e tomadas ações a, $Q^*(s, a) = \max_{\pi} E\,[R_t \mid s_t = s, a_t = a,\ \pi]$, onde π é definida como uma política que mapeia sequências de estados para ações.

A função de ação-valor ótima $Q^*(s, a)$ obedece a identidade conhecida como Equação de Bellman. Baseado na seguinte intuição: se o valor ótimo $Q^*(s', a')$ da sequência s' no próximo *timestep* é conhecida para todas as ações a', então a estratégia ótima será selecionar a ação a'que maximiza o valor de retorno esperado de $r + \gamma Q^*(s', a')$.

$$Q^*(s,a) = E_{s'\sim\epsilon}\left[r + \gamma \max_{a'} Q^*(s',a') \mid s,a\right] \quad (2)$$

A ideia básica de muitos algoritmos de RL é a de estimar a função de ação valor utilizando a equação de Bellman como uma atualização iterativa, $Q^*(s,a) = E\left[r + \gamma \max_{a'} Q^*(s',a') \mid s,a\right]$. Estes algoritmos de iteração de valores convergem para a função de valor ótima, $Q_i \to Q^*$ conforme $i \to \infty$ (SUTTON; BARTO, 2018). No entanto, este raciocínio não é prático, pois, a função de ação-valor é estimada separadamente para cada sequência, o que a torna incapaz de generalizar para outros casos. Por conta disso, é comum utilizar um aproximador de funções de ação-valor, $Q(s,a;\theta) \approx Q^*(s,a)$. Em RL este aproximador é tipicamente um aproximador de funções lineares, mas, em alguns casos um aproximador de funções não lineares é utilizado, tal como redes neurais. Refere-se a uma rede neural como aproximador de funções com pesos θ como *Q-network*. Assim, uma rede neural pode ser treinada para minimizar uma sequência de funções de *loss* $L_i(\theta_i)$ que muda a cada nova iteração *i*.

$$L_i(\theta_i) = E_{s,a\sim\rho(s,a)}\left[\left(y_i - Q(s,a;\theta_i)\right)^2\right] \quad (3)$$

Onde $y_i = E_{s'\sim\epsilon}[r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) \mid s,\ a]$ é o alvo para a iteração *i* e $\rho(s,a)$ são uma distribuição de probabilidades sobre sequências de estados s e ações a as quais são denominadas distribuição de comportamento. Os parâmetros da iteração anterior θ_{i-1} são mantidos fixos quando a função de *loss* $L_i(\theta_i)$ está sendo otimizada. Note que, os alvos dependem dos pesos da rede, isso contrasta com os alvos utilizadas para aprendizado supervisionado, que são fixos antes do aprendizado iniciar. Derivando-se a função de *loss* em respeito aos pesos, chega-se ao seguinte gradiente.

$$\nabla_{\theta_i} L_i(\theta_i) = E_{s,a\sim\rho(s,a);\, s'\sim\epsilon}\left[\left(r + \gamma \max Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i)\right) \nabla_{\theta_i} Q(s,a;\theta_i)\right] \quad (4)$$

Contrário a se calcular todas as expectativas no gradiente acima, é conveniente otimizar-se a função de *loss* por um gradiente estocástico. Note que este algoritmo é *model-free*, isto é, resolve o problema utilizando somente amostras do ambiente ϵ, sem necessariamente construir uma aproximação dele. Também, este algoritmo é *off-policy*, ou seja, este aprende sobre a estratégia *greedy* $a = \max_a Q(s,a;\ \theta)$ enquanto segue uma distribuição de comportamentos que garante exploração adequada do espaço de estados.

Tabela 2 – Pseudocódigo do algoritmo DQN

Deep Q-*learning* with Experience Replay
Inicializar replay de memória D para capacidade N
Inicializar função de ação-valor Q com pesos randômicos (rede neural)
Para episódio = 1, M faça:
Inicializar sequência $s_1 = \{x_1\}$ e sequência pré-processada $\varphi_1 = \varphi(s_1)$
Iterar para cada *step t* = 1, T do episódio:
Com probabilidade ∈ selecionar uma ação aleatoriamente a_t
Senão, selecionar $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
Executar uma ação a_t e receber recompensa r_t e uma distância ao alvo d
Configura o próximo estado $s_{t+1} = s_t, a_t$ e executa o pré-processamento da função $\phi_{t+1} = \phi(s_{t+1})$
Guarda a transição $(\phi_t, a_t, r_t, \phi_{t+1})$ na memória D
Pega amostras de pequenos lotes de transição da memória D
Configura $y_j = \begin{cases} r_j & para\ \phi_{j+1}\ terminal \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & para\ \phi_{j+1}\ não{-}terminal \end{cases}$
Executa o passo da descida do gradiente em $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ de acordo com a equação (4).
Fim iteração T
Fim iteração M

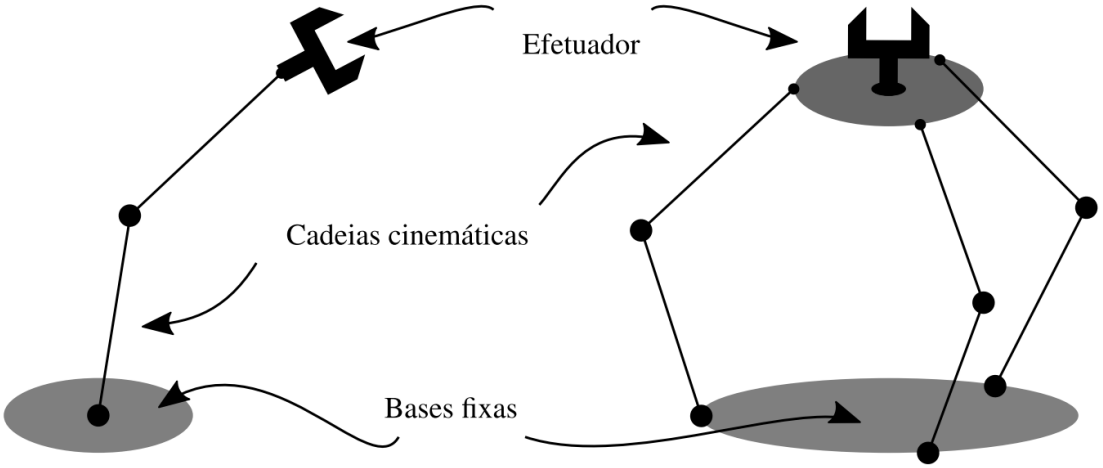
Fonte: Adaptado de (MNIH; et. al, 2013).

2.5 ROBÔS PARALELOS

Os robôs manipuladores industriais são em geral classificados em dois tipos: robôs seriais e robôs paralelos (TAGHIRAD, 2013). Os robôs industriais mais conhecidos são os robôs manipuladores seriais, que são empregados no mundo todo principalmente em linhas de montagens. São assim denominados devido ao fato de suas juntas e *links* estarem dispostas de maneira "serial", ou seja, normalmente há uma base fixa da qual um *link* se conecta com uma junta e assim sequencialmente até o efetuador do robô (TSAI, 1999). Esta sequência de *links* e juntas é denominada de cadeia cinemática. Nas linhas de produção do setor automotivo robôs seriais são utilizados para funções de soldagem, rebitagem e montagem, dentre outras. Existem ainda aplicações destes na logística em empresas de empacotamento e distribuição, ou em qualquer aplicação onde se requer um robô capaz de operar as mais diversas cargas com um espaço de trabalho relativamente grande os mecanismos de natureza serial geralmente são uma

boa solução. A Figura 3 apresenta um comparativo dentre os mecanismos de natureza serial e os de natureza paralela.

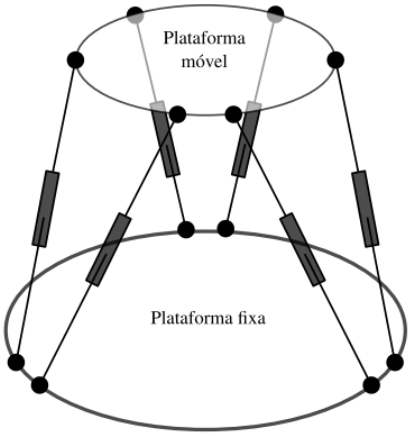
Figura 3 - Comparação Robôs Seriais e Robôs Paralelos



Fonte: Adaptado de YESHMUKHAMETOV, 2017.

Robôs paralelos possuem cadeias cinemáticas dispostas paralelamente uma à outra e, onde estas são unidas por uma plataforma fixa e uma plataforma móvel. Sendo que a plataforma móvel é geralmente o efetuador do mecanismo. Dentre estes mecanismos, pode-se destacar a plataforma de Stewart-Gough, que foi originalmente concebida com objetivo de testar a performance de pneus tal como demonstrado por Gough e Whitehall (1962). O mesmo mecanismo foi utilizado para aplicações em simuladores de voo no trabalho desenvolvido por Stewart (1965). A Figura 4 apresenta um esquemático da plataforma.

Figura 4 - Plataforma de Stewart-Gough



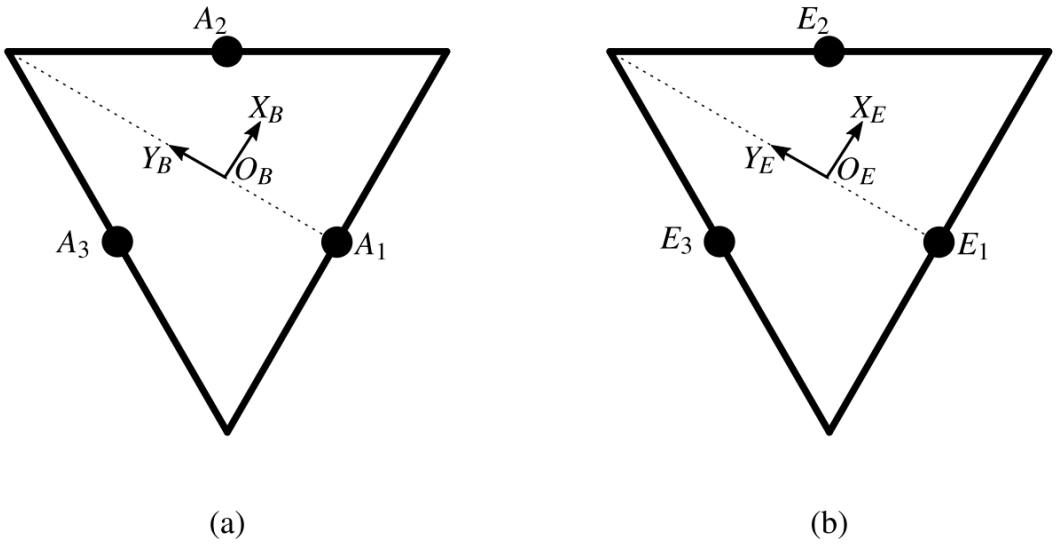
Fonte: Adaptado de CHI et al, 2015.

2.6 ROBÔ DELTA

O robô Delta é um mecanismo paralelo com três graus de liberdade translacionais. Em termos práticos é capaz de posicionar o seu efetuador no espaço sem rotacionar o mesmo (WILLIAMS, 2016).

A Figura 5 (a) mostra a vista superior da base fixa do robô, observa-se que com o sistema de referência escolhido, o ponto A_1 está na direção negativa do eixo Y_B, esta convenção foi adotada, pois, facilitará os cálculos posteriores. Observa-se na Figura 5 (b) a escolha de orientação de eixos para o centro da plataforma móvel e o ponto E_1.

Figura 5 - Plataformas fixa (a) e móvel (b)



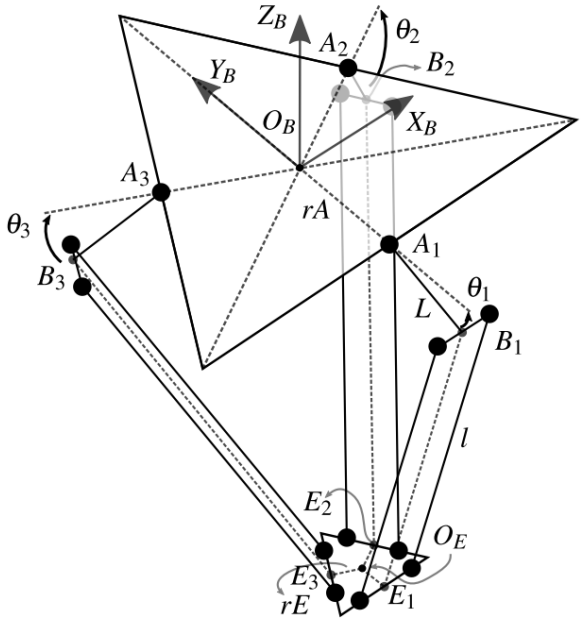
Fonte: Autor.

Na Figura 6 tem-se o esquema mecânico completo do robô Delta, tem-se os pontos de origem da plataforma fixa O_B, os pontos A_1, A_2 e A_3 nos quais os atuadores rotativos são montados. Além disso, observa-se um tipo de mecanismo especial, os paralelogramos $\overrightarrow{B_iE_i}$. Os pontos E_1, E_2 e E_3 se encontram no centro do lado do triângulo que forma a plataforma móvel, o centro desta é o ponto de interesse O_E para o efetuador do robô. Observa-se também os parâmetros construtivos mais importantes do robô Delta, que são os seguintes.

- rA - Distância do centro da plataforma fixa e a extremidade mais próxima do triângulo.
- L - Comprimento do braço do atuador.
- l - Comprimento do paralelogramo.

- rE - Distância do centro da plataforma móvel (efetuador) até a extremidade mais próxima do triângulo.

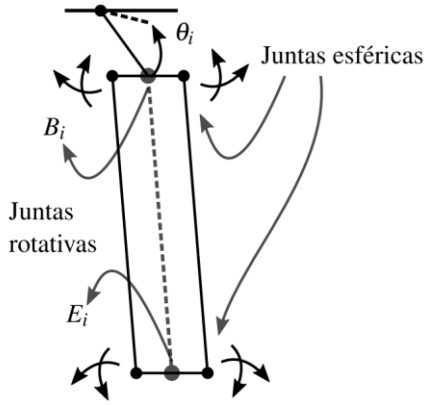
Figura 6 - Mecanismo completo do robô Delta



Fonte: Adaptado de ZAVATSKY (2009).

A Figura 7 mostra como o paralelogramo consegue manter o efetuador sempre paralelo à plataforma fixa. Os paralelogramos são a chave que garante uma alta velocidade para o robô Delta devido à sua alta rigidez mecânica e movimento puramente translacional. Estes fazem com que a plataforma móvel sempre esteja orientada paralelamente à plataforma fixa. Um importante componente do paralelogramo são as juntas, pode-se optar pelas juntas esféricas nas partes mais externas, ou mesmo utilizar juntas rotativas no lugar das esféricas pois para o mecanismo são utilizadas somente as rotações indicadas na Figura 7. Estas são executadas nos eixos X e Y sendo uma destas executada pelo ponto B_i. Por fim, pode-se observar que os pontos B_i e E_i são juntas rotativas.

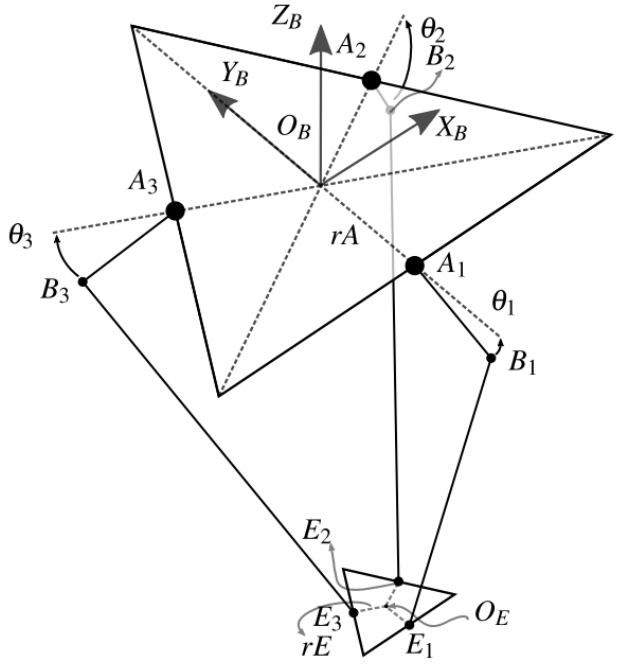
Figura 7 - Vista lateral do paralelogramo



Fonte: Autor.

A Figura 8 mostra uma versão simplificada do robô Delta, esta foi utilizada para se determinar as cinemáticas inversa e direta.

Figura 8 - Mecanismo do Robô Delta simplificado



Fonte: Adaptado de ZAVATSKY (2009).

2.7 ESTUDO DA CINEMÁTICA DOS MECANISMOS DO ROBÔ DELTA

Para a cinemática inversa do Delta, tem-se que dadas as coordenadas para o efetuador $E_O = [X_E, \quad Y_E, \quad Z_E]$ deseja-se determinar os ângulos dos atuadores $\Theta = [\theta_1, \quad \theta_2, \quad \theta_3]$ para que o efetuador se posicione no ponto especificado. Este ângulo é medido a partir da base da plataforma fixa e este é positivo no sentido negativo de Z. Ou seja, sempre que um ou mais pontos $B_i$estiver acima da plataforma fixa, tem-se um ou mais ângulos negativos. Ao se tratar

de cinemática direta deseja-se determinar o ponto no espaço que um dado conjunto de ângulos dos atuadores determina.

2.7.1 Cinemática Inversa

Como observado anteriormente cinemática inversa busca resolver os ângulos $\Theta = [\theta_1, \ \theta_2, \ \theta_3]$ para um ponto de entrada do efetuador $E_O = [X_E, \ Y_E, \ Z_E]$. No entanto, inicialmente deve-se observar uma característica importante da geometria do robô Delta, o fato de que, cada um de seus atuadores está posicionado a 120° com relação ao outro, portanto, o primeiro passo na resolução da cinemática inversa é a rotação do ponto inicial no eixo *Z* em 120° e −120°.

$$E_1 = \begin{bmatrix} E_{X_1} \\ E_{Y_1} \\ E_{Z_1} \end{bmatrix} \quad E_2 = \begin{bmatrix} E_{X_2} \\ E_{Y_2} \\ E_{Z_2} \end{bmatrix} \quad E_3 = \begin{bmatrix} E_{X_3} \\ E_{Y_3} \\ E_{Z_3} \end{bmatrix} \tag{5}$$

A matriz de rotação pura em *Z* é determinada da seguinte maneira:

$$R_Z(\phi) = \begin{bmatrix} cos(\phi) & -sin(\phi) & 0 \\ sin(\phi) & cos(\phi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{6}$$

Portanto, rotacionou-se E_2em $\phi = -120°$ e E_3em $\phi = 120°$. Outro fator de simplificação é a coordenada *Z* do ponto já que essa é a mesma para todos e que a rotação não causa nenhuma alteração, assim pode-se utilizar uma mesma variável para todos $E_{Z_1} = E_{Z_2} = E_{Z_3} = E_Z$.

$$E_2 = R_Z(-120).E_1 \tag{7}$$

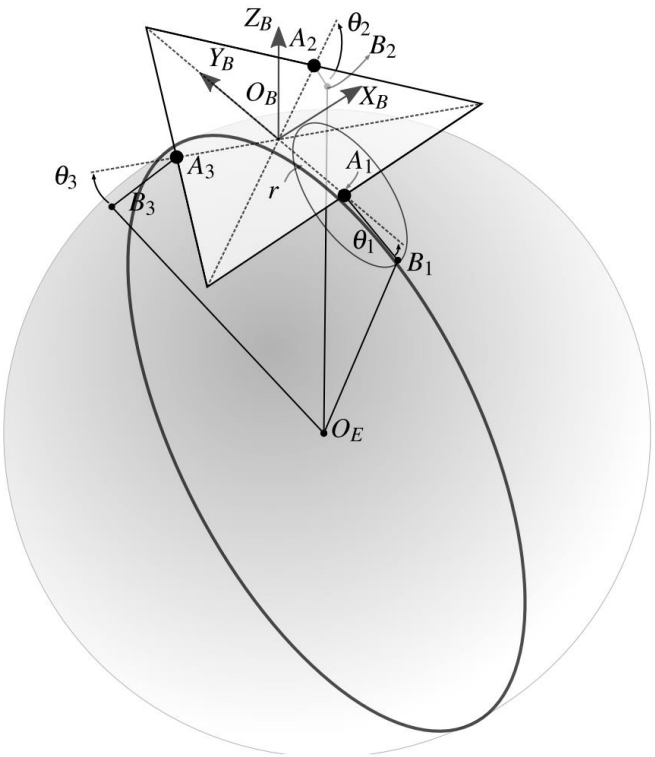
$$E_2 = \begin{bmatrix} E_{X_1}.cos(-120) - E_{Y_1}.sin(-120) \\ E_{X_1}.sin(-120) + E_{Y_1}.cos(-120) \\ E_Z \end{bmatrix} \tag{8}$$

Seguindo este mesmo raciocínio, as coordenadas do ponto E_3foram deduzidas:

$$E_3 = \begin{bmatrix} E_{X_1}.cos(120) - E_{Y_1}.sin(120) \\ E_{X_1}.sin(120) + E_{Y_1}.cos(120) \\ E_Z \end{bmatrix} \tag{9}$$

Simplificou-se os raios das plataformas para tornar o efetuador um ponto único no espaço. A Figura 9 representa esta simplificação. Além disso, demonstra o princípio no qual é baseada a solução, cinemática para o robô Delta, a intersecção entre esferas e círculos.

Figura 9 - Mecanismo do robô Delta com intersecção de círculo e esfera

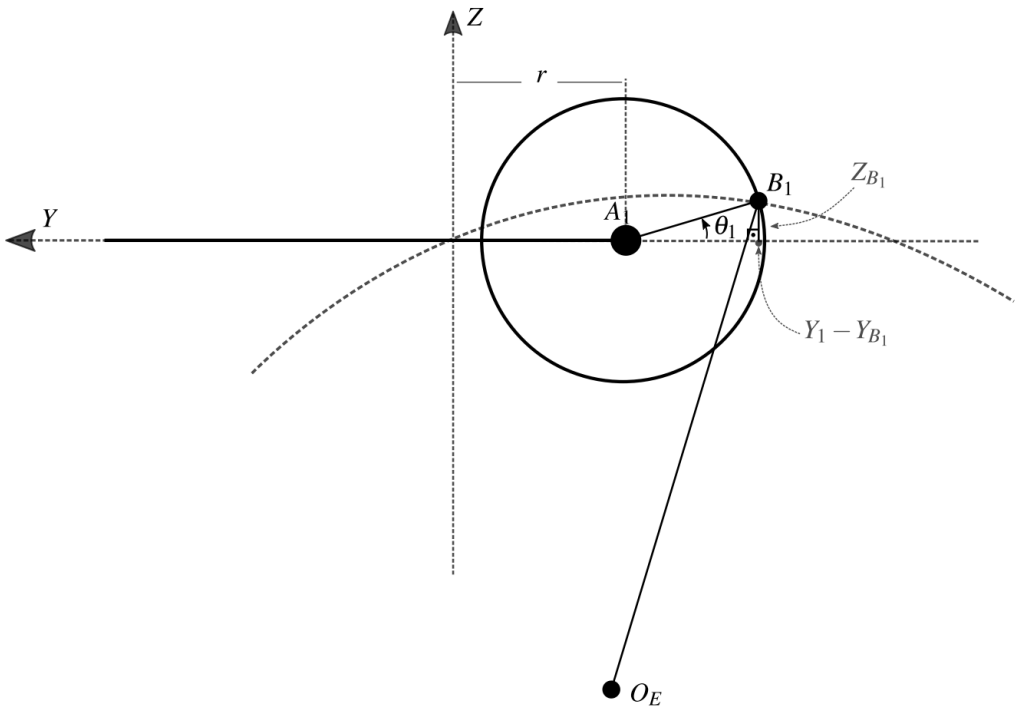


Fonte: Adaptado de ZAVATSKY (2009).

Destaca-se que o ponto A_1é o centro de um círculo cujo raio é o comprimento do braço do atuador, além disso, este círculo se intersecta com uma esfera cujo centro é o ponto do efetuador O_E e cujo raio é o comprimento do paralelogramo destes pontos, este valor é o módulo da distância dos pontos$B_i E_i$, ou seja, um valor fixo. Os paralelogramos foram removidos para que fosse possível observar com clareza as esferas e os círculos formados.

Na Figura 10 observa-se uma visão lateral do mecanismo, nos eixos Z × Y.

Figura 10 - Visão do mecanismo no plano Z × Y



Fonte: Autor.

O valor de $r$que define a simplificação dos raios das plataformas nada mais ´e que o valor negativo de rA raio da plataforma fixa, somado ao raio da plataforma móvel rE. Esta simplificação é possível pois a plataforma só executa movimentos translacionais, ou seja, não existem rotações nos movimentos do robô (STAMPER et al., 1997).

$$r = -rA + rB \tag{10}$$

Foi utilizado o valor negativo por ter sido utilizada a orientação dos atuadores no valor negativo de Y. A distância entre os pontos A_ie B_ié um parâmetro construtivo, assim como a distância dos pontos B_ieE_i. Foram definidas as seguintes variáveis para tais distâncias

$$L = \overrightarrow{A_\iota B_\iota} \quad l = \overrightarrow{B_\iota E_\iota} \tag{11}$$

Com estas definições foram determinadas as equações de círculo e esfera. A equação (11) é a equação do círculo centrado no ponto A_i. Devido a escolha de orientação, a determinação da coordenada do centro do círculo se dá pela diferença entre a coordenada do raio simplificado re o valor de coordenadas dos pontos B_i para a cadeia escolhida, não há coordenada para X.

$$L^2 = (-r + B_{Y_i})^2 + (-B_{Z_i})^2 \tag{12}$$

Para a equação da esfera centrada em O_E com raio l tem-se:

$$l^2 = (X_{E_i})^2 + (Y_{E_i} - B_{Y_i})^2 + (Z_{E_i} - B_{Z_i})^2 \tag{13}$$

Subtrai-se a equação (11) da equação (12) resultando na equação seguinte:

$$B_{Z_i} = \left(\frac{r - Y_{E_i}}{Z_{E_i}}\right) B_{Y_i} + \left(\frac{L^2 - l^2 + {X_{E_i}}^2 + {Y_{E_i}}^2 + {Z_{E_i}}^2 - r^2}{2Z_{E_i}}\right) \tag{14}$$

Para facilitar as operações, substitui-se os valores entre parênteses por duas variáveis auxiliares a_ieb_i. Esta equação é então reduzida a apenas:

$$B_{Z_i} = a_i B_{Y_i} + b_i \tag{15}$$

Onde

$$a_i = \left(\frac{r - Y_{E_i}}{Z_{E_i}}\right) \qquad b_i = \left(\frac{L^2 - l^2 + {X_{E_i}}^2 + {Y_{E_i}}^2 + {Z_{E_i}}^2 - r^2}{2Z_{E_i}}\right) \tag{16}$$

Observou-se que a equação (13) representa a relação das variáveis, porém, foi necessária uma outra substituição para que a equação pudesse ser solucionada. Assim, isolou-se a variável B_{Z_i}da equação (11) e foi feita a substituição da mesma na equação (13) da qual foi obtida uma equação do segundo grau com solução pela equação de Bhaskara:

$${B_{Y_i}}^2 + \left(\frac{2a_i b_i - 2r}{1 + {a_i}^2}\right) B_{Y_i} + \left(\frac{r^2 + {b_i}^2 - L^2}{1 + {a_i}^2}\right) = 0 \tag{17}$$

As soluções da equação quadrática de segundo grau possuem a seguinte solução. Dada uma equação do segundo grau no seguinte formato:

$$ax^2 + bx + c = 0 \tag{18}$$

Sua solução é dada pela seguinte equação:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{19}$$

Ao se resolver para a incógnita B_{Y_i} basta que se substitua a mesma na equação (15) para obter B_{Z_i}. Assim, ao aplicar Bhaskara na equação (17), obtém-se:

$$B_{Y_i} = -\left(\frac{a_i b_i - r}{1 + {a_i}^2}\right) \pm \frac{1}{2}\sqrt{\left(\frac{2a_i b_i - 2r}{1 + {a_i}^2}\right)^2 - 4\left(\frac{r^2 + {b_i}^2 - L^2}{1 + {a_i}^2}\right)} \tag{20}$$

$$B_{Y_i} = -\left(\frac{a_i b_i - r}{1 + {a_i}^2}\right) \pm \frac{1}{2}\sqrt{\left(\frac{2a_i b_i - 2r}{1 + {a_i}^2}\right)^2 - 4\left(\frac{r^2 + {b_i}^2 - L^2}{1 + {a_i}^2}\right)} \tag{21}$$

Por fim, utiliza-se a seguinte relação trigonométrica para aferir-se o ângulo do atuador:

$$\theta_i = a\,sin\left(\frac{-B_{Z_i}}{L}\right) \tag{22}$$

2.7.2 Cinemática Direta

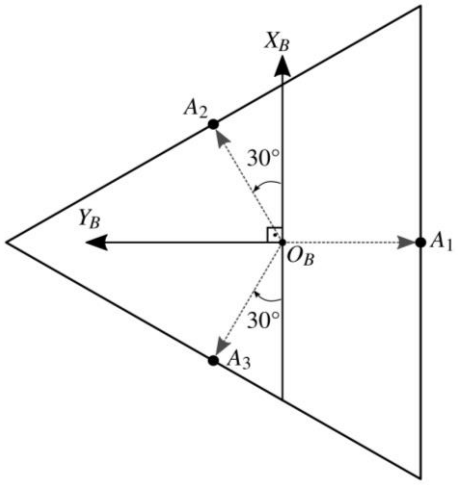
Para a cinemática direta, tem-se como objetivo encontrar o ponto $E_O = [X_E, \quad Y_E, \quad Z_E]$ dados ângulos $\Theta = [\theta_1, \quad \theta_2, \quad \theta_3]$. Inicia-se com a formalização das coordenadas dos pontos dos atuadoresA_1, A_2eA_3. Com auxílio da Figura 10 e Figura 11, foi possível obter tais coordenadas. A mesma simplificação dos raios das plataformas foi utilizada para a cinemática direta, no entanto, de acordo com a literatura o sinal de r é invertido, ou seja, temos que $r = rA - rE$ devido a escolha de orientação para a plataforma.

Foram obtidas as seguintes coordenadas para a localização dos atuadores A_1, A_2eA_3:

$$A_1 = \begin{bmatrix} 0 \\ -(r + L\,cos(\theta_1)) \\ -L\,cos(\theta_1) \end{bmatrix} \qquad A_2 = \begin{bmatrix} (r + L\,cos(\theta_2))\,cos(\phi) \\ (r + L\,cos(\theta_2))\,sin(\phi) \\ -L\,cos(\theta_1) \end{bmatrix}$$
$$A_3 = \begin{bmatrix} -(r + L\,cos(\theta_3))\,cos(\phi) \\ (r + L\,cos(\theta_3))\,sin(\phi) \\ -L\,cos(\theta_3) \end{bmatrix} \tag{23}$$

Onde $\phi = 30°$, de acordo com a Figura 11. A Figura 12 representa as esferas que o movimento dos braços descreve no espaço, a intersecção destas resulta no ponto E_O. Além disso na Figura 12 demonstra que o centro das esferas fica localizado nos pontosB_i:

Figura 11 - Vista superior da base fixa em detalhes



Fonte: Autor.

Para se determinar as coordenadas do atuador deve-se resolver as equações para as três esferas descritas na Figura 12.

As equações de esfera centradas nos pontos B_1, B_2eB_3, são, respectivamente:

$$l^2 = (X_E - B_{X_1})^2 + (Y_E - B_{Y_1})^2 + (Z_E - B_{Z_1})^2 \quad (24)$$

$$l^2 = (X_E - B_{X_2})^2 + (Y_E - B_{Y_2})^2 + (Z_E - B_{Z_2})^2 \quad (25)$$

$$l^2 = (X_E - B_{X_3})^2 + (Y_E - B_{Y_3})^2 + (Z_E - B_{Z_3})^2 \quad (26)$$

Ao se expandir as equações acima descritas, foi deduzida uma simplificação visando um menor custo computacional, da seguinte forma:

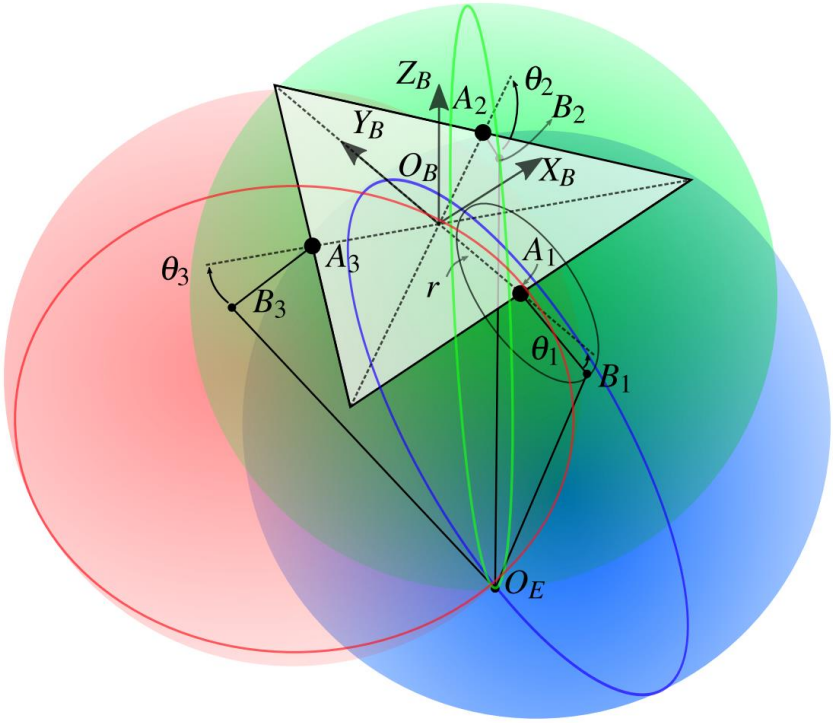
$$\rho_i = {B_{X_i}}^2 + {B_{Y_i}}^2 + {B_{Z_i}}^2 \quad (27)$$

$${X_E}^2 + {Y_E}^2 + {Z_E}^2 - 2B_{X_1}X_E - 2B_{Y_1}Y_E - 2B_{Z_1}Z_E = l^2 - \rho_1 \quad (28)$$

$${X_E}^2 + {Y_E}^2 + {Z_E}^2 - 2B_{X_2}X_E - 2B_{Y_2}Y_E - 2B_{Z_2}Z_E = l^2 - \rho_2 \quad (29)$$

$$X_E{}^2 + Y_E{}^2 + Z_E{}^2 - 2B_{X_3}X_E - 2B_{Y_3}Y_E - 2B_{Z_3}Z_E = l^2 - \rho_3 \quad (30)$$

Figura 12 - Vista das esferas dos braços do robô Delta



Fonte: Adaptado de ZAVATSKY (2009).

Subtraiu-se as equações (31) da (32), a equação (31) da (33) e a equação (32) da (33). O que resultou nas seguintes expressões:

$$(B_{X_1} - B_{X_2})X_E + (B_{Y_1} - B_{Y_2})Y_E + (B_{Z_1} - B_{Z_2})Z_E = \frac{\rho_1 - \rho_2}{2} \quad (31)$$

$$(B_{X_1} - B_{X_3})X_E + (B_{Y_1} - B_{Y_3})Y_E + (B_{Z_1} - B_{Z_3})Z_E = \frac{\rho_1 - \rho_3}{2} \quad (32)$$

$$(B_{X_2} - B_{X_3})X_E + (B_{Y_2} - B_{Y_3})Y_E + (B_{Z_2} - B_{Z_3})Z_E = \frac{\rho_2 - \rho_3}{2} \quad (33)$$

Isolando-se Y_E da equação (32) obteve-se:

$$Y_E = \frac{\frac{(\rho_1 - \rho_3)}{2} - (B_{X_1} - B_{X_3})X_E - (B_{Z_1} - B_{Z_3})Z_E}{(B_{Y_1} - B_{Y_3})} \quad (34)$$

Substituiu-se a equação (34) na equação (33) o que determinou a relação entre X_E e $Z_E$com o seguinte formato:

$$X_E = a_1 Z_E + b_1 \tag{35}$$

As constantes a_1 e b_1 serviram para reduzir a complexidade do equacionamento, estas foram definidas do seguinte modo:

$$a_1 = \frac{1}{d}\left[\left(B_{Z_2} - B_{Z_1}\right)\left(B_{Y_3} - B_{Y_1}\right) - \left(B_{Z_3} - B_{Z_1}\right)\left(B_{Y_2} - B_{Y_1}\right)\right] \tag{36}$$

$$b_1 = \frac{1}{2d}\left[(\rho_2 - \rho_1)\left(B_{Y_3} - B_{Y_1}\right) - (\rho_3 - \rho_1)\left(B_{Y_2} - B_{Y_1}\right)\right] \tag{37}$$

A constante auxiliar d foi determinada da seguinte maneira:

$$d = (B_{X_2} - B_{X_1})(B_{Y_3} - B_{Y_1}) - (B_{X_3} - B_{X_1})(B_{Y_2} - B_{Y_1}) \tag{38}$$

O mesmo procedimento foi utilizado para obter-se Y_E e Z_E:

$$Y_E = a_2 Z_E + b_2 \tag{39}$$

As constantes a_2e b_2 foram descritas da seguinte forma:

$$a_2 = \frac{1}{d}\left[\left(B_{Z_2} - B_{Z_1}\right)\left(B_{Y_3} - B_{Y_1}\right) - \left(B_{Z_3} - B_{Z_1}\right)\left(B_{Y_2} - B_{Y_1}\right)\right] \tag{40}$$

$$b_2 = \frac{1}{2d}\left[(\rho_2 - \rho_1)\left(B_{Y_3} - B_{Y_1}\right) - (\rho_3 - \rho_1)\left(B_{Y_2} - B_{Y_1}\right)\right] \tag{41}$$

Com estas relações pode-se resolver para a variável Z_E de acordo com a equação quadrática do segundo grau abaixo:

$$\begin{aligned}(a_1{}^2 + a_2{}^2 + 1){Z_E}^2 + 2\left(a_1 + a_2\left(b_2 - B_{Y_1}\right) - B_{Z_1}\right)Z_E \\ + \left({b_1}^2 + \left(b_2 - B_{Y_1}\right)^2 + {B_{Z_1}}^2 - l^2\right) = 0\end{aligned} \tag{42}$$

Assim conclui-se o estudo teórico do robô Delta. A solução aqui descrita não é a única possível, pois, existem muitas outras interpretações deste mesmo problema.

3 MATERIAIS E MÉTODOS

Nesse capítulo serão apresentados os *softwares* e linguagens de programação utilizadas para o desenvolvimento do trabalho. O trabalho foi desenvolvido inteiramente na linguagem de programação *Python*®, que foi utilizado por conta de sua grande quantidade de bibliotecas que facilitam o desenvolvimento de projetos de aprendizado máquina.

Para implementação do algoritmo DQN, desenvolveu-se um ambiente com base nas equações de cinemática direta do robô Delta, tal como demonstrado no item 4.1. O ambiente foi projetado utilizando-se a linguagem de programação *Python*® e as bibliotecas *matplotlib* e *numpy*. O algoritmo implementado foi baseado no trabalho de Mnih e colaboradores (2015), no entanto, com algumas adaptações e para o ambiente do robô Delta, foi possível resolver o ambiente em menos de 2000 episódios como será visto a seguir.

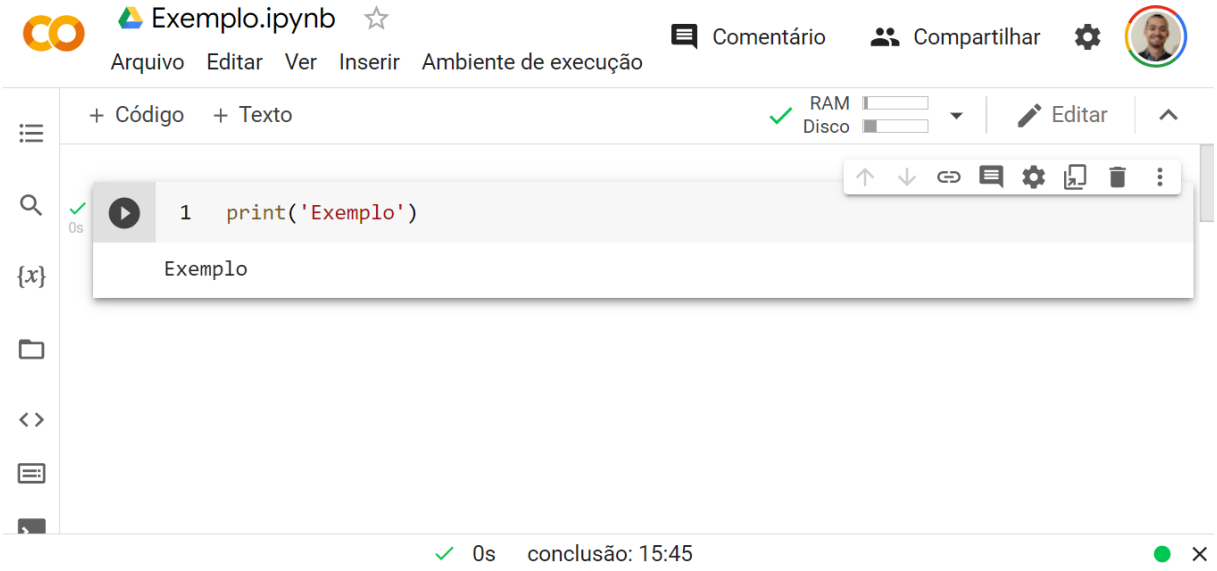
3.1 AMBIENTE DE DESENVOLVIMENTO

Para desenvolvimento e execução dos algoritmos tanto de treinamento quanto de exibição do ambiente utilizou-se o *Google Colaboratory*® que é um ambiente de desenvolvimento *online*. A empresa Google® disponibiliza hardware de ponta para a execução dos algoritmos de aprendizado de máquina que notadamente exigem o uso significativo de processamento por GPU por conta do uso dos tensores da biblioteca PyTorch.

Para utilização do ambiente *online* basta efetuar o login em uma conta do Gmail, o ambiente funciona com estrutura de Jupyter Notebooks, o qual se pode dividir por células, nessas células pode-se executar scripts do *Python*® e os resultados são exibidos na parte inferior destas. Essa maneira de se trabalhar se tornou muito utilizada por Cientistas de Dados que precisam muitas vezes trabalhar com grandes *datasets* que podem exceder a memória de um computador. Assim, ao se trabalhar com uma plataforma *online* reduz-se drasticamente o impacto na performance da execução dos algoritmos permitindo que a atenção do usuário fique no problema em que está resolvendo e não em limitações de *hardware*.

A Figura 14 apresenta o ambiente de desenvolvimento, note a execução de uma função simples do Python a qual imprime o valor textual passado, é possível adicionar mais células como e as executar junto, ou de maneira separada, assim o desenvolvimento nessa plataforma é intuitivo, fácil e acessível, o que permitiu o foco no projeto a ser implementado.

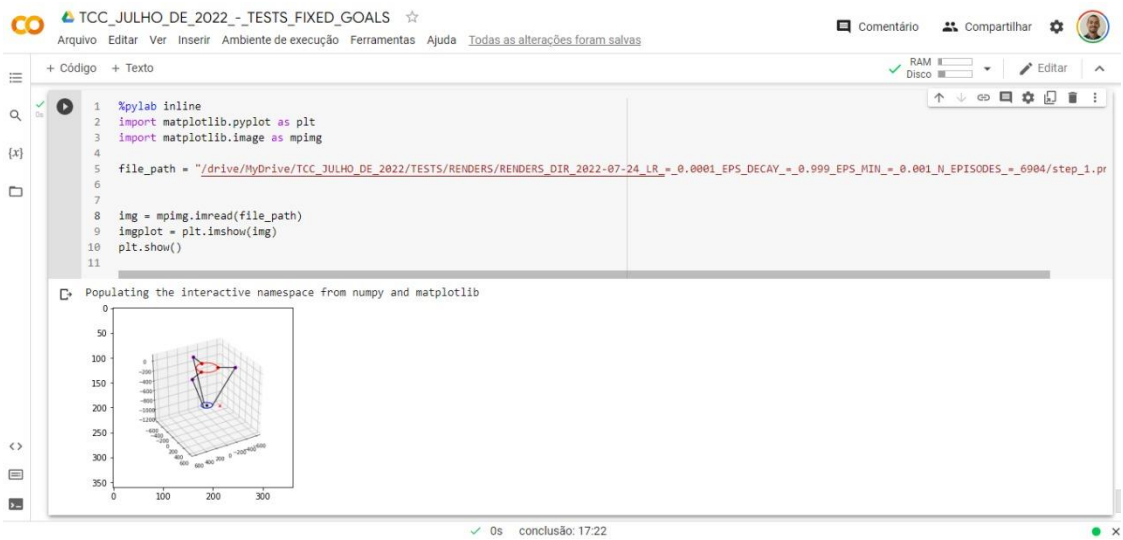
Figura 13 - Plataforma de Desenvolvimento Google Colaboratory



Fonte: Autor.

Para que fosse possível observar o robô foi desenvolvido um algoritmo que utiliza a cinemática direta do mecanismo para desenhar o robô na tela a cada novo *timestep*, isso foi possível pelo fato de que o algoritmo implementado resolve somente a cinemática do robô, isto é, a dinâmica da plataforma não faz parte do escopo deste projeto. Dessa forma, em cada *timestep* uma nova figura do ambiente é gerada e salva em um local específico para análise posterior. A Figura 14 apresenta como se pode renderizar o ambiente do robô a partir de dados salvos anteriormente.

Figura 14 - Robô Delta no ambiente de desenvolvimento



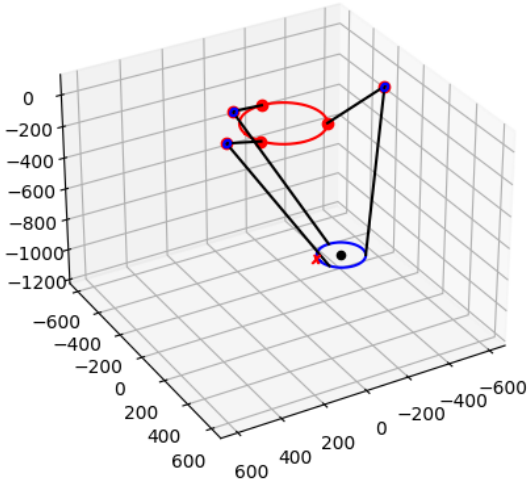
Fonte: Autor.

3.2 DESCRIÇÃO DO AMBIENTE DO ROBÔ DELTA

O ambiente desenvolvido tem como escopo demonstrar o funcionamento do algoritmo em um robô paralelo Delta, sendo que este não aborda a dinâmica do mecanismo, somente a sua cinemática. O objetivo do agente é determinar os ângulos dos atuadores do robô de forma que seu efetuador se posicione em um ponto aleatório no espaço tridimensional, ou seja, dado um ponto no espaço (x, y, z), o agente deve escolher sequência de ações ótimas (que retornam a maior recompensa) para os atuadores com a finalidade de posicionar o efetuador do robô neste ponto.

No ambiente o alvo é representado pela letra X na cor vermelho, o efetuador do robô é representado por um ponto preto no centro da plataforma móvel, enquanto a plataforma fixa é vermelha e os atuadores são pontos na mesma cor localizados nos vértices dela. A Figura 15 mostra um episódio de uma execução do algoritmo. O ambiente foi implementado utilizando a linguagem de programação *Python*®, a biblioteca *matplotlib* foi utilizada para apresentar o mecanismo do robô Delta, como na Figura 18.

Figura 15 - Ambiente do robô Delta



Fonte: Autor.

3.2.1 Ações no ambiente do robô Delta

A tabela abaixo apresenta todas as ações possíveis para o ambiente desenvolvido. A ação 0, significa manter todas as juntas na mesma posição. A ação 1 significa aumentar o ângulo da junta 1, e assim por diante, com combinações de incrementos e decrementos para cada junta do robô, formando assim um conjunto de 15 ações possíveis.

Tabela 3 - Ações do agente no ambiente Delta

0	HOLD
1	INC_J1
2	DEC_J1
3	INC_J2
4	DEC_J2
5	INC_J3
6	DEC_J3
7	INC_J1_J2
8	DEC_J1_J2
9	INC_J2_J3
10	DEC_J2_J3
11	INC_J1_J3
12	DEC_J1_J3
13	INC_J1_J2_J3
14	DEC_J1_J2_J3

Fonte: Autor.

Com esse conjunto de ações o agente foi capaz de resolver a tarefa do ambiente, não foi necessário adicionar um conjunto mais complexo de ações.

3.2.2 Estados no ambiente do robô Delta

Os estados formam um conjunto de dados relacionados às posições do efetuador do robô, os ângulos das juntas, posição do alvo e a distância entre o efetuador e o objetivo. Tal como a Tabela 4 abaixo.

Para o cálculo da distância entre o efetuador e o objetivo, utilizou-se a diferença da norma de cada vetor posição. Na equação (40) observa-se a norma do vetor posição do ponto objetivo, enquanto na equação (41) tem-se a norma para o vetor posição do efetuador. Para se calcular a distância basta fazer-se o módulo da diferença das duas, tal como na equação (42).

$$\left\|\vec{v}_{obj}\right\| = \sqrt{x_{obj}^2 + y_{obj}^2 + z_{obj}^2} \tag{43}$$

$$\|\vec{v}_{ee}\| = \sqrt{x_{ee}^2 + y_{ee}^2 + z_{ee}^2} \tag{44}$$

$$d = \sqrt{(x_{obj} - x_{ee})^2 + (y_{obj} - y_{ee})^2 + (z_{obj} - z_{ee})^2} \tag{45}$$

Na Tabela 4 temos os estados para o ambiente desenvolvido.

Tabela 4 - Tabela de estados do ambiente

Posição do Alvo	Posição do Efetuador	Ângulos dos atuadores	Distância do efetuador ao alvo
x_{obj}	x_{ee}	θ_1	d
y_{obj}	y_{ee}	θ_2	
z_{obj}	z_{ee}	θ_3	

Fonte: Autor.

3.3 DEFINIÇÕES DE PROPRIEDADES FÍSICAS DO MECANISMO

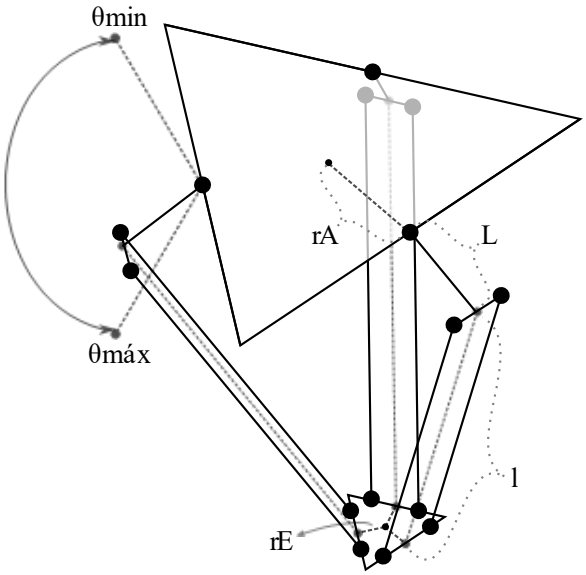
Em termos de definições de propriedades físicas utilizou-se um modelo comercial o robô paralelo Delta modelo FANUC® FANUC-M3iA Figura 16, cujas dimensões estão demonstradas na

Figura 17.

Figura 16 - Robô Delta industrial

Fonte:

Figura 17 - Dimensões robô FANUC-M3iA



Fonte: Autor.

A Tabela 5 apresenta as principais dimensões do modelo de robô Delta utilizado.

Tabela 5 - Dimensões físicas do robô

Dimensão	Valor [mm]
rA	180
L	400
l	900
θ_{min}	-55°
θ_{max}	90°
Z_{Emin}	-585

Fonte: Autor.

3.4 DESCRIÇÃO DA ARQUITETURA DE REDE UTILIZADA E IMPLEMENTAÇÃO DO ALGORTIMO DQN

Para o aprendizado profundo aplicado neste trabalho utilizou-se uma arquitetura de rede Perceptron multicamadas. A rede é constituída de oito camadas das quais seis são camadas escondidas. A Tabela 6 mostra como estas camadas estão dispostas. Tal como na Tabela 3, observe que na última camada da rede temos quinze ações.

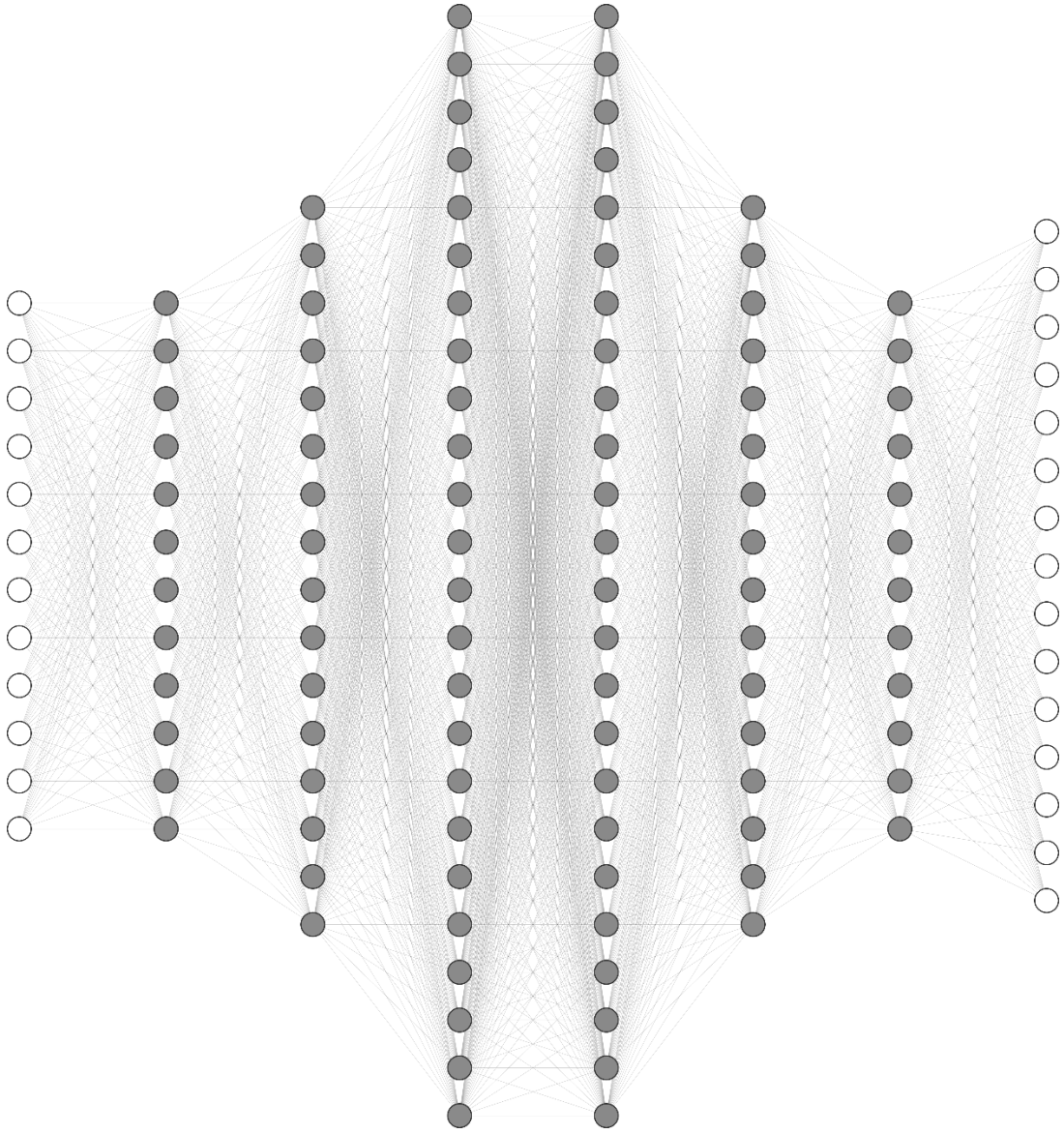
Tabela 6 – Descrição da disposição dos neurônios artificiais na rede

Estados	Camadas escondidas da rede						Ações
10	300	400	600	600	400	300	15

Fonte: Autor.

A Figura 18 apresenta a rede neural completa, nela pode-se observar as camadas de entrada e saída neurônios de cor branca, e as camadas escondidas que estão representados pela cor mais escura. As camadas escondidas estão em escala, isto é, cada neurônio escuro equivale à 25 neurônios reais da aplicação.

Figura 18 - Representação da rede neural utilizada

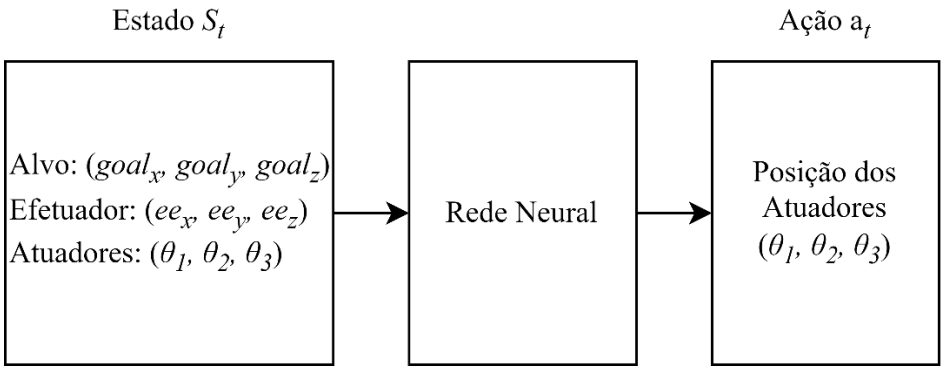


Fonte: Autor.

O algoritmo foi implementado utilizando a biblioteca *PyTorch*® do *Python*® a qual é atualmente muito utilizada para implementação de algoritmos de Aprendizado por Reforço. O otimizador utilizado foi o algoritmo Adam (KINGMA; BA, 2015) já implementado na biblioteca e para medição do erro utilizou-se a função *Mean Squared Error* também disponível na mesma.

O algoritmo funciona de forma iterativa, recebendo como entrada os estados do ambiente, e como saída este escolhe uma ação entre quinze ações possíveis no ambiente, que estão descritas na Tabela 3. Para melhor visualizar o fluxo de iterações a Figura 19 apresenta os estados de entrada na rede e a saída desta.

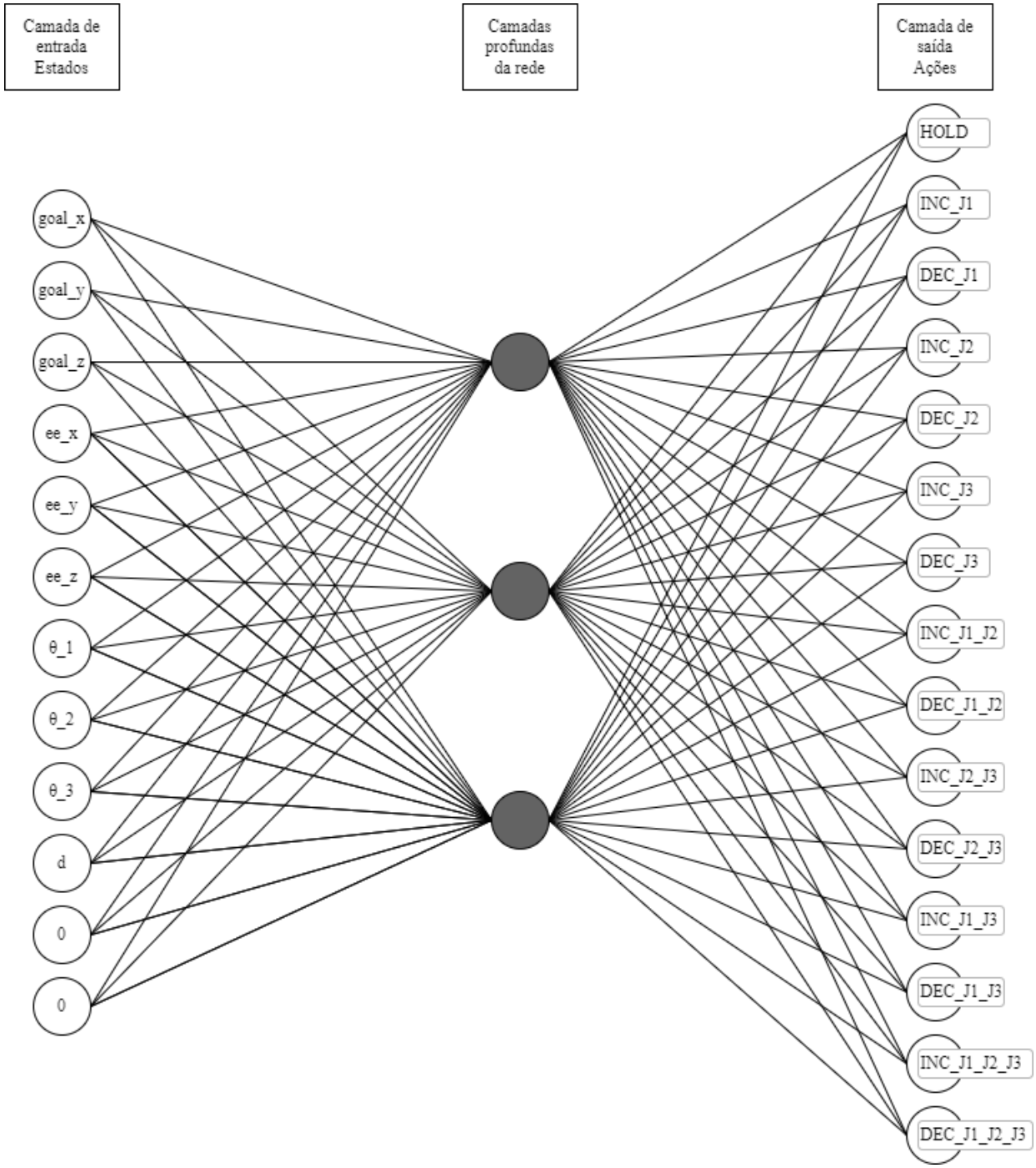
Figura 19 - Fluxo iterativo do agente no ambiente



Fonte: Autor.

A Figura 20 apresenta a arquitetura de rede proposta, note que na camada de entrada existem dois neurônios com valor fixo em zero, isto é necessário por conta do tamanho das matrizes, para que os cálculos sejam possíveis, estas devem ter o *shape* adequado.

Figura 20 - Arquitetura de Rede Neural



Fonte: Autor.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

Onde:

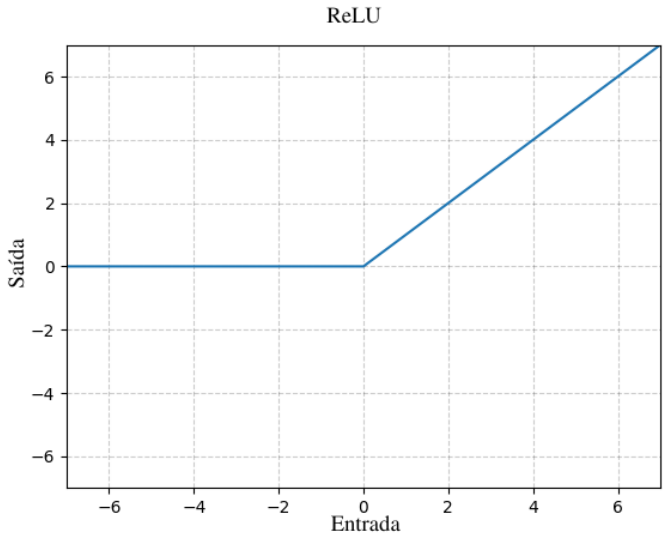
- MSE - *Mean Squared Error*
- n – Quantidade de pontos
- Y_i – Valores observados
- $\hat{Y}_i$ – Valores previstos

A função de ativação utilizada na rede foi a ReLU (Rectified Linear Unit) é uma função de ativação que retorna sempre os valores positivos de seu argumento e, é definida matematicamente da seguinte forma.

$$f(x) = \begin{cases} 0, & x < 0 \\ x, & x \geq 0 \end{cases}$$

A Figura 21 apresenta o gráfico da função de ativação utilizada no algoritmo.

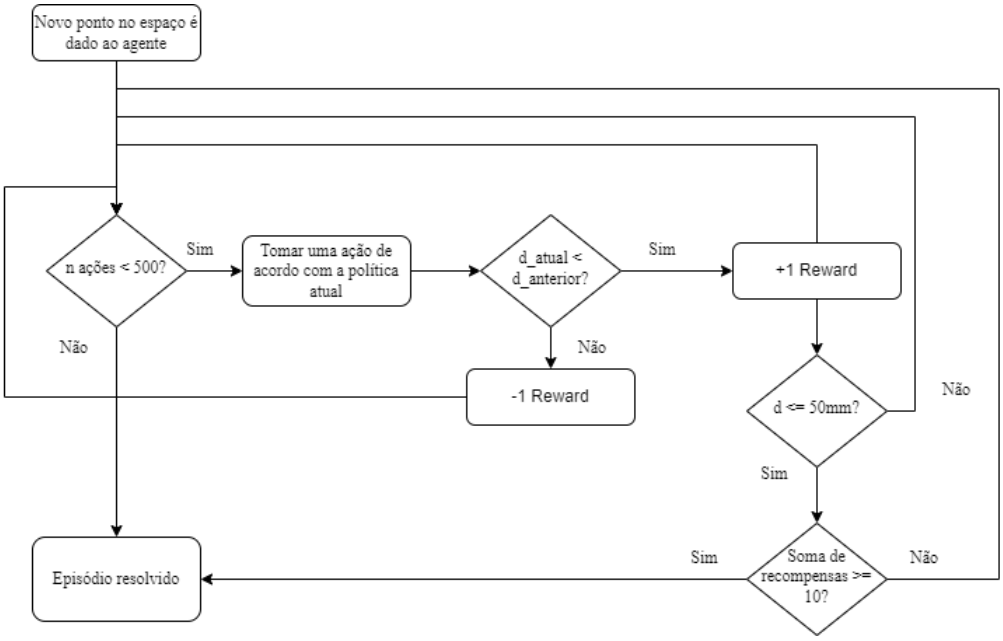
Figura 21 - Função de Ativação ReLU



Fonte: Adaptado de pytorch.org.

A Figura 22 apresenta o fluxograma do algoritmo, observe que a cada novo ponto recebido pelo agente, uma série de cálculos ocorre para que seja possível se tomar uma ação com base nos estados do robô.

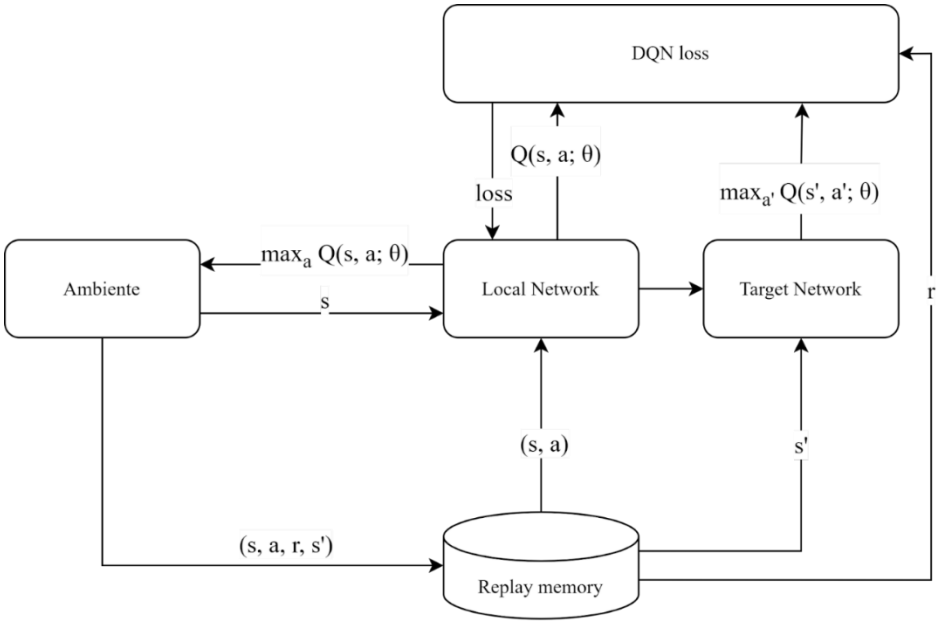
Figura 22 - Fluxograma do algoritmo



Fonte: Autor.

A Figura 23 apresenta o diagrama de funcionamento interno ao algoritmo, como se pode observar, duas redes neurais são utilizadas durante o treinamento, a rede determinada *Local Network* é utilizada para explorar o ambiente e executar as ações enquanto a rede determinada como *Target Network* é utilizada para selecionar as ações que retornam as maiores recompensas.

Figura 23 - Fluxograma interno do Algoritmo DQN



Fonte: NAIR, A., 2015

4 RESULTADOS

Os resultados obtidos para o trabalho foram suficientes para alcançar os objetivos que motivaram o seu desenvolvimento. A seguir, os resultados são demonstrados.

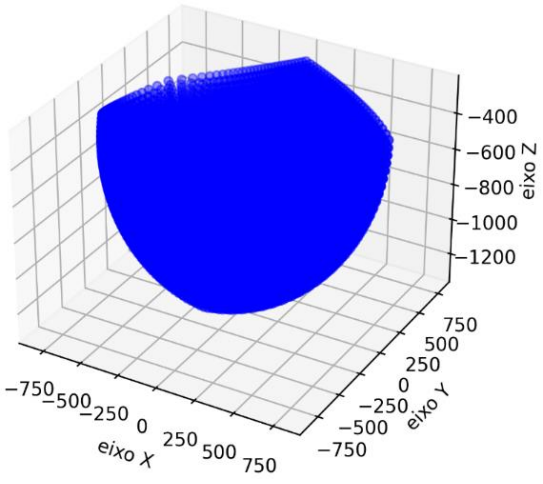
4.1 RESULTADOS DAS DEFINIÇÕES DE CINEMÁTICA DIRETA E INVERSA

A partir das definições de Cinemática Direta e Inversa foram obtidos os seguintes resultados, o volume de trabalho e trajetórias típicas de robótica implementadas no robô Delta.

4.1.1 Volume de Trabalho do Robô Delta

O volume de trabalho determina os pontos no espaço nos quais o robô é capaz de se posicionar. Trata-se de uma informação muito importante especialmente para fabricantes que produzem robôs para a indústria. O volume de trabalho é usado para determinar os pontos em que o robô pode alcançar além de ser possível entender algumas limitações de acordo com a configuração do mecanismo. Na Figura 22 o volume de trabalho do robô é demonstrado, os eixos X e Y representam o plano do espaço onde o ponto de coordenada (0, 0) é o centro da posição do efetuador do robô, o eixo Z é o eixo de altura do robô, este apresenta valores negativos pois para a modelagem matemática considera-se a base fixa do robô como estando no ponto de coordenadas (0, 0, 0).

Figura 24 - Volume de Trabalho robô Delta



Fonte: Autor.

4.1.2 Trajetórias Típicas aplicadas ao robô Delta

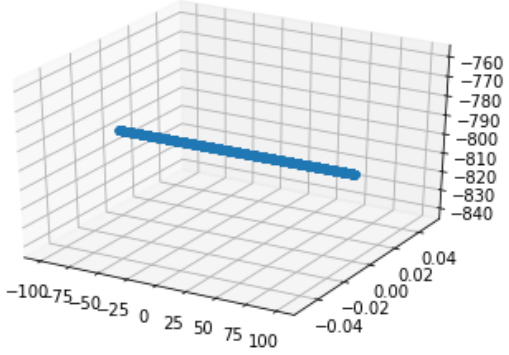
Nessa subseção serão abordadas as trajetórias que foram implementadas utilizando as equações de cinemática Inversa e Direta do robô Delta. Para que fossem obtidas tais trajetórias

inicialmente foram determinadas funções que pudessem gerar as sequências de pontos. Posteriormente a sequência foi passada como entrada para a função de cinemática inversa do robô, com isso, o algoritmo retorna uma lista de coordenadas dos atuadores do robô que posicionam o seu efetuador nas coordenadas dos pontos de entrada.

4.1.2.1 Trajetória Retilínea

Para a trajetória retilínea, a biblioteca *numpy* foi utilizada para gerar um vetor com 10000 pontos, este foi utilizado para a variável X enquanto as variáveis Y e Z foram mantidas constantes sendo $Y = 0$ e $Z = -800$ mm. Dessa forma, a trajetória da Figura 25 foi obtida.

Figura 25 - Trajetória retilínea robô Delta

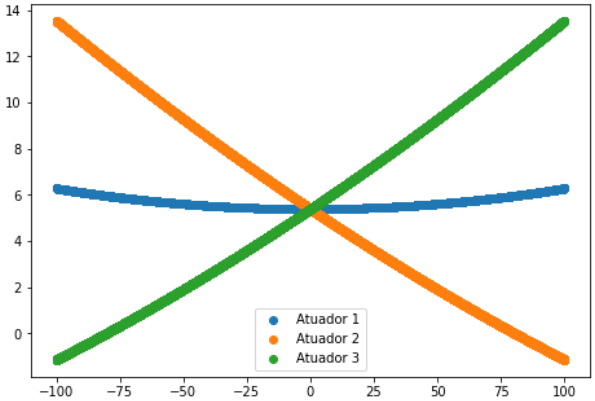


Fonte: Autor.

Os pontos se encontram em coordenadas do espaço tridimensional (X, Y e Z), trata-se dos pontos nos quais se deseja que o efetuador do robô se posicione no espaço.

Na Figura 26 estão descritos os ângulos dos atuadores ao longo da trajetória retilínia, os eixos X, e Y desta representam respectivamente um vetor de valores variando de -100 a 100 no eixo X qual os ângulos dos atuadores, eixo Z estão sendo projetados, dessa forma, é possível se analisar o comportamento do ângulo dos atuadores ao performarem a trajetória retilínea.

Figura 26 - Ângulos resultantes da trajetória retilínea



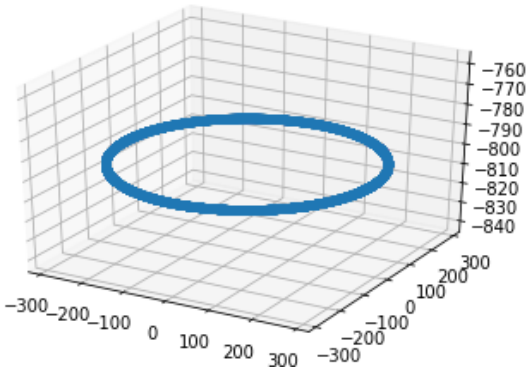
Fonte: Autor.

Os pontos na Figura 26 se encontram no espaço de juntas do robô, ou seja, trata-se de coordenadas dos ângulos nos quais os atuadores devem se posicionar ao longo do tempo para que o efetuador do mesma descreva a trajetória desejada.

4.1.2.2 Trajetória Circular

Foi descrita uma trajetória circular com raio de 300 mm fixado na altura de -800 mm em relação a base fixa do robô. A Figura 26 apresenta uma trajetória circular, onde os eixos X e Y são o plano e o eixo Z a altura do efetuador, note que neste último, os valores são negativos tendo em vista que a base fixa do robô fica na coordenada Z = 0.

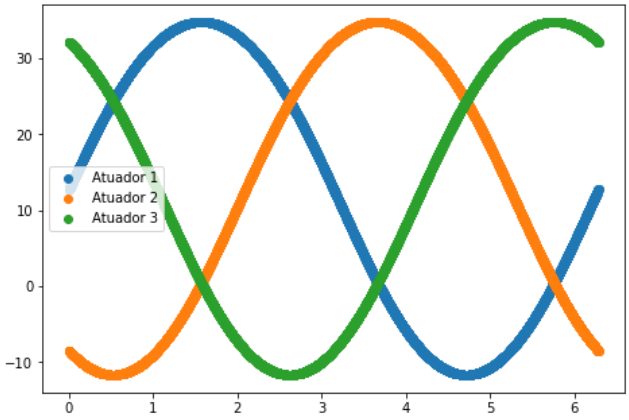
Figura 27 - Trajetória circular



Fonte: Autor.

Os ângulos dos atuadores estão descritos na Figura 28 no espaço de juntas. Nessa Figura o eixo Y representa o valor do ângulo em graus e o eixo X apresenta um vetor de valores que variam de 0 até 2π, dessa forma os eixos permitem a correta exibição do comportamento dos ângulos no espaço.

Figura 28 - Ângulos dos atuadores para a trajetória circular

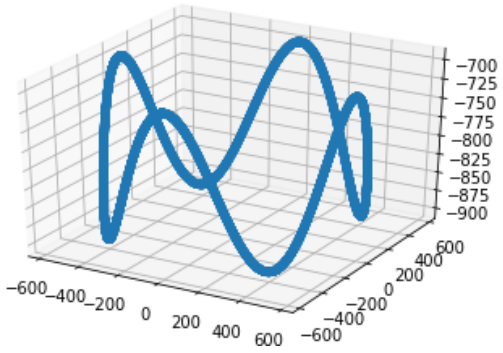


Fonte: Autor.

4.1.2.3 Trajetória Circular com Senoide em Z

A Figura 29 mostra uma trajetória um pouco mais complexa na qual foi determinada uma equação que descreve um círculo no plano X vs. Y e uma senoide no plano Z. Eixos X e Y da Figura representam o plano no espaço, note os valores negativos em Z devido a definição inicial do sistema de coordenadas.

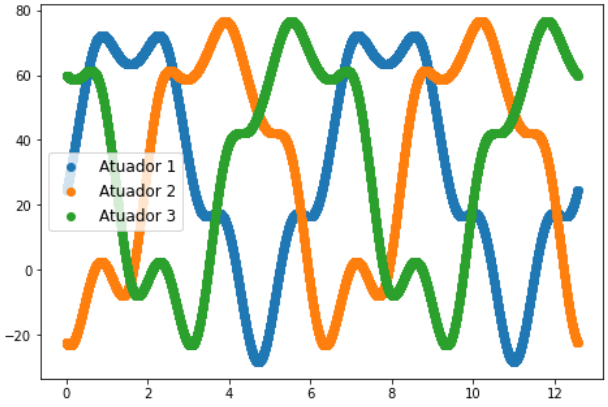
Figura 29 – Trajetória circular com senoide em Z



Fonte: Autor.

A Figura 28 apresenta como os atuadores se comportam no espaço de juntas do robô para a trajetória em questão. O eixo Y mostra os valores dos ângulos em graus enquanto que o eixo X é um vetor de valores que começa em 0 e vai até 4π.

Figura 30 - Trajetória no espaço de juntas com a senoide em Z e círculo em X vs. Y.

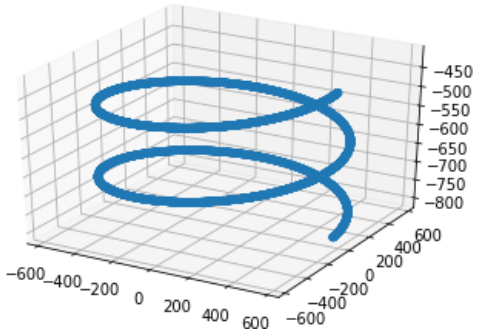


Fonte: Autor.

4.1.2.4 Trajetória espiral em Z

Na Figura 31 pode-se observar uma trajetória em espiral descendente em Z. Os eixos X e Y da Figura determinam o plano e o eixo Z representa a altura do efetuador do robô.

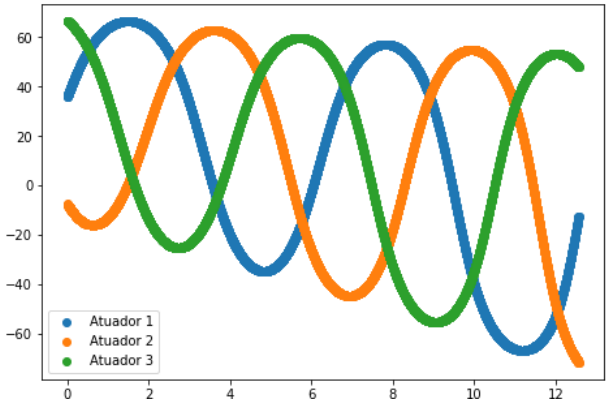
Figura 31 - Trajetória em espiral



Fonte: Autor.

A Figura 32 apresenta a posição angular dos atuadores no espaço de juntas do mesmo durante a trajetória espiral em Z. Nessa Figura tem-se os valores dos ângulos dos atuadores no eixo Y, enquanto que no eixo X tem-se um vetor de valores que variam de 0 até 4π.

Figura 32 - Posição dos atuadores para a trajetória em espiral

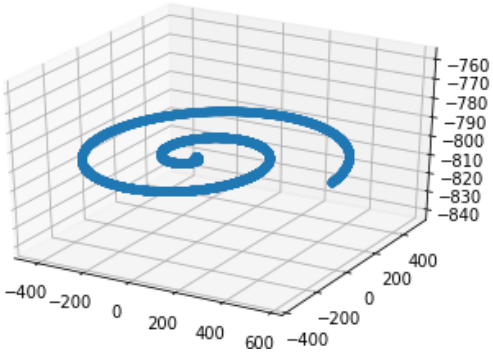


Fonte: Autor.

4.1.2.5 Trajetória espiral no plano X vs. Y

A Figura 33 apresenta a trajetória espiral no plano X vs. Y descrita pelo efetuador do robô Delta. Os eixos X e Y da Figura representam o plano enquanto o eixo Z representa a altura do efetuador, o valor de Z para essa trajetória foi mantido fixo em -800.

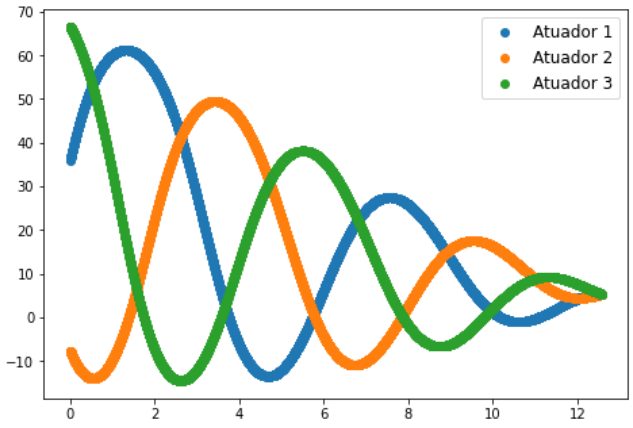
Figura 33 - Espiral no plano X vs. Y



Fonte: Autor.

Na Figura 34 temos a trajetória dos atuadores ao performar a trajetória do efetuador da Figura 33. A Figura 32 mostra os ângulos dos atuadores no eixo Y enquanto o eixo X é um vetor de valores que vão de 0 até 4π.

Figura 34 - Trajetória dos atuadores do robô no espaço de juntas

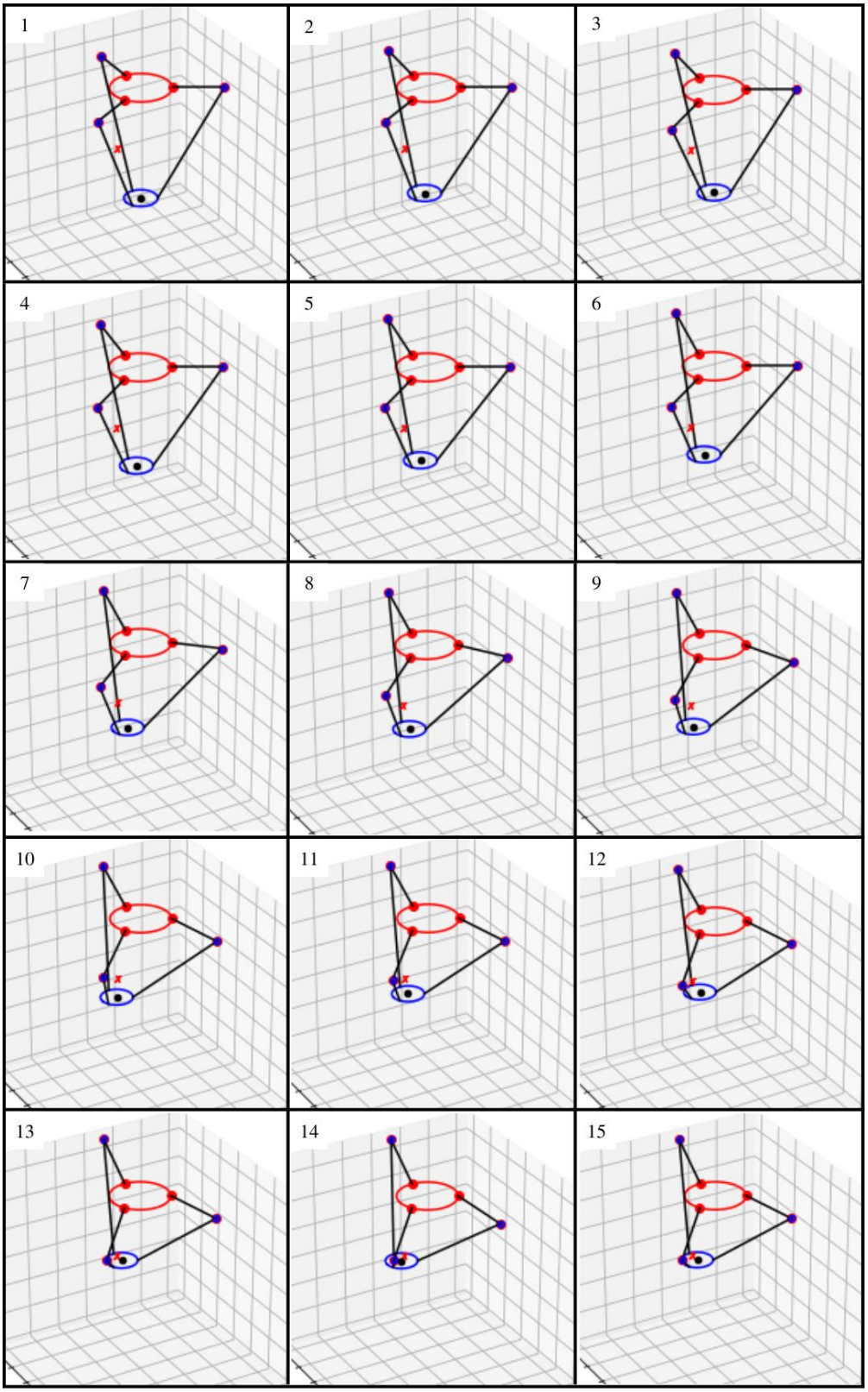


Fonte: Autor.

4.2 RESULTADOS DO ALGORITMO DQN

Como pode-se observar o agente conseguiu resolver o ambiente para os pontos fixos aos quais foi submetido. Na Figura 35 é possível observar o robô no ambiente durante execução de um episódio que ele resolve em quinze *timesteps*. O robô deve posicionar o seu efetuador no ponto x representado nas figuras, que são uma sequência de ações tomadas pelo robô durante o episódio em questão.

Figura 35 - Episódio completo no ambiente



Fonte: Autor.

Os resultados obtidos ao executar o algoritmo DQN no ambiente desenvolvido foram suficientes para resolver o problema em questão. O critério de aceitação para solução do problema é o que segue: em um número predefinido de episódios e passos o algoritmo deve posicionar corretamente o seu efetuador em um ponto no espaço de trabalho do robô. O algoritmo foi configurado de tal forma que o robô deve manter seu efetuador a uma distância máxima de 50 milímetros do ponto no espaço, o movimento do robô segue a modelagem matemática previamente estabelecida para ele, acumulando assim as recompensas necessárias para validar suas ações a cada *timestep*.

Para que fosse possível observar os dados do ambiente, foram estabelecidos os estados do robô conforme a Tabela 4. Assim, em cada novo passo dado o ambiente muda, retornando um novo estado.

Os estados terminais de cada episódio são dois, o episódio termina quando:

1. O número de passos previsto terminou.
2. O ambiente foi resolvido.

4.2.1 Parâmetros e configurações utilizadas

Um dos parâmetros mais importantes para o algoritmo DQN é o épsilon, pois a escolha correta deste e de seu decaimento vai garantir a correta convergência para a solução dentro de um número adequado de episódios calculados com base neste.

Durante os testes percebeu-se que com o parâmetro errado, o algoritmo convergia, no entanto, logo em seguida perdia performance, fenômeno observado em outros trabalhos no qual o algoritmo DQN pode “esquecer” do seu aprendizado, tal problema foi solucionado na literatura com a implementação do algoritmo *Prioritized Replay Buffer* o qual é capaz de priorizar na memória do agente nos episódios em que este obteve uma melhor performance (SCHAUL T.; et al.).

Para os resultados aqui apresentados, o cálculo adequado do número de episódios para o épsilon foi suficiente para o algoritmo resolver o ambiente. A metodologia utilizada para abordar o dilema *exploration* vs. *exploitation* foi o de decaimento de *epsilon greedy*. Em outras palavras o parâmetro decai ao longo do treinamento, assim, o agente inicia explorando o ambiente com diversas políticas diferentes e com o passar dos episódios este cada vez mais dá preferência às melhores políticas aprendidas (daí o *greedy* do inglês “ganancioso”) visando obter o máximo possível de recompensas positivas.

Para o cálculo do épsilon literatura define a seguinte equação.

$$\epsilon_{max} \cdot \epsilon^{x}_{decay} = \epsilon_{min}$$

Onde definiu-se que:

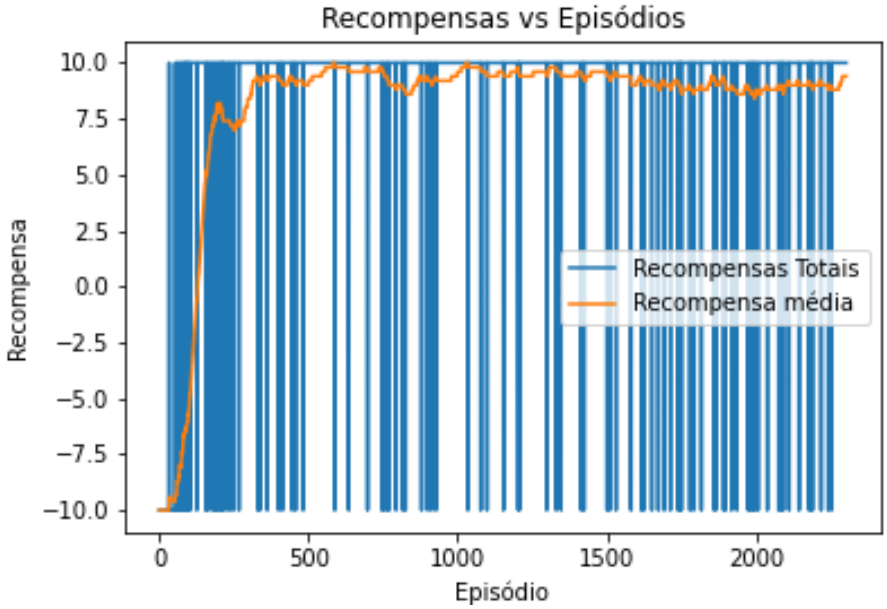
- $\epsilon_{max} = 0.1$
- $\epsilon_{decay} = 0.999$
- $\epsilon_{min} = 0.01$

Para os parâmetros definidos acima, foi obtido $x = 2302$ que é o número de episódios utilizado no treinamento e nos testes com pontos aleatórios.

4.2.2 Resultados obtidos durante o treinamento

A Figura 36 apresenta o gráfico de recompensas vs. episódios ao rodar o treinamento do agente no ambiente. O cálculo da curva de média é feito com base nos últimos cem episódios, por conta disso, está apresenta valores menores nos primeiros episódios.

Figura 36 - Recompensas durante o treinamento



Fonte: Autor.

Para cada episódio o agente pode tomar até quinhentas ações no ambiente, acumulando recompensas ao final destes, em cada episódio a recompensa máxima é 10, enquanto a punição máxima é -10. Dessa forma, o agente desenvolveu uma política de ações através da qual acertou 94% dos episódios durante o treinamento. A Tabela 7 apresenta o número total de episódios e o número de episódios em que o agente obteve o número máximo de recompensas.

Tabela 7 - Número de acertos no treinamento

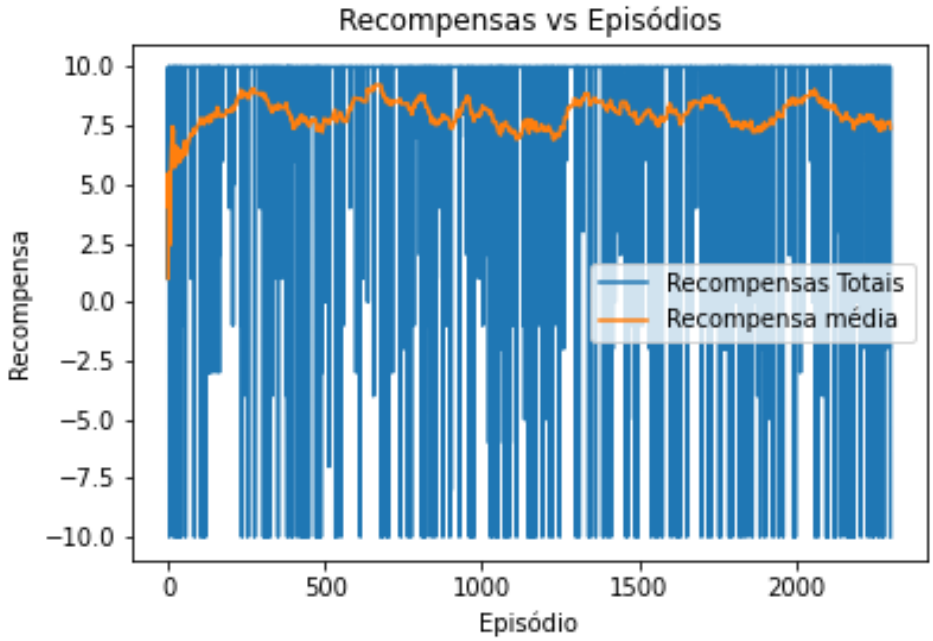
Número de episódios	Número de acertos
2302	2165

Fonte: Autor.

4.2.3 Resultados de testes com pontos aleatórios no ambiente

Um dos testes realizados foi o de carregar a rede neural no agente e apresentar para estes pontos aleatórios em seu ambiente. A Figura 36 apresenta o gráfico de episódio vs. recompensa durante os testes realizados, para estes o ambiente foi configurado de tal forma a gerar pontos aleatórios no espaço de trabalho do robô. Observou-se que nesse teste o agente acertou 85% dos episódios.

Figura 37 - Recompensas durante os testes



Fonte: Autor.

Como é possível observar na Figura 36, os resultados obtidos durante os testes com pontos aleatórios mostram que o algoritmo foi capaz de generalizar para pontos gerados fora de seu treinamento, isto é, resolveu estados com uma política de ações adequada tanto para pontos conhecidos quanto para pontos novos. A Tabela 8 apresenta o número de acertos do agente nos testes com pontos aleatórios.

Tabela 8 - Número de acertos durante testes com pontos aleatórios

Número de episódios	Número de acertos
2302	1928

Fonte: Autor.

4.2.4 Resultados de testes com pontos fixos no ambiente

Nos experimentos com pontos fixos observou-se que as limitações do modelo da cinemática inversa do robô se refletem também no comportamento do agente quando apresentado a situações extremas do mecanismo em questão. Isto é, quando um dado ponto no espaço está localizado próximo de regiões de singularidade – termo usado para designar pontos no espaço nos quais o controle e precisão do mecanismo ficam com limitações – o agente se mostra incapaz de resolver o ambiente, refletindo em uma performance pobre tal como demonstram dados obtidos durante testes.

Tabela 9 – Testes com pontos fixos no ambiente

Experimento	Ângulos dos atuadores (graus°)	Coordenadas do efetuador (mm)	Número de tentativas	Número de acertos
1	[-45, -45, -45]	[0, 0, -541]	1000	1000
2	[0, -45, -45]	[0, 225, -559]	1000	1000
3	[0, 0, -45]	[-235, 136, -613]	1000	1000
4	[45, -45, 0]	[245, 453, -572]	1000	1000
5	[30, 30, 0]	[-223, 129, -872]	1000	1000
6	[-45, 25, -25]	[-254, -268,-575]	1000	1000
7	[-10, 30, -30]	[-354, -72, -654]	1000	1000
8	[45, 45, 45]	[0, -0,-1106]	1000	942
9	[-30, -30, 0]	[149, -86,-620]	1000	1000
10	[-45, 30, -30]	[-303, -266, -559]	1000	1000
11	[30, 0, 45]	[349, 52, -878]	1000	1000
12	[15, -15, 25]	[269, 73, -772]	1000	969
13	[10,-25, -15]	[56, 195, -669]	1000	1000
14	[-35, -15, 5]	[105, -184, -636]	1000	1000
15	[10, -30, 10]	[237, 137, -687]	1000	1000
16	[-25, 10, -5]	[-88, -185, -689]	1000	1000
17	[-30, 5, 30]	[160, -333, -681]	1000	1000
18	[-30, 45, 30]	[-114, -497, -690]	1000	1000
19	[-15, 10, 15]	[33, -205, -757]	1000	1000
20	[-5, -10, 5]	[92, -18, -733]	1000	1000
21	[-10, 30, 40]	[82, -371, -821]	1000	1000

Fonte: Autor.

A Tabela 9 apresenta os dados obtidos ao testar-se o agente em pontos fixos conhecidos no espaço, a primeira coluna representa o número do episódio, a segunda os ângulos dos atuadores para o ponto escolhido, a terceira coluna representa a coordenada do ponto no espaço, já a quarta coluna apresenta o número de vezes que o mesmo episódio foi estado, por último, a quinta coluna apresenta quantos desses episódios o agente conseguiu resolver corretamente.

5 CONCLUSÃO

Se pode concluir, portanto, que os objetivos do trabalho foram alcançados conforme determinado no escopo inicial deste.

A modelagem matemática do mecanismo se deu pela derivação das equações de cinemática Direta e Inversa da estrutura do robô. Para se comprovar a correta implementação das equações obteve-se o espaço de trabalho do efetuador do robô.

Com a derivação das equações foi implementado um ambiente de simulação utilizando-se da cinemática direta para plotagem da estrutura do robô bem como do seu alvo, tornando possível observar graficamente o robô no ambiente de aprendizado proposto.

Utilizando-se de referências de outros autores e bibliotecas de desenvolvimento disponíveis para Python foi implementado um algoritmo DQN ao ambiente desenvolvido, através deste algoritmo o agente foi capaz de aprender e determinar com precisão acima de 90% e alguns casos o ponto determinado para sua posição no ambiente.

Com o algoritmo implementado e o agente treinado no ambiente foram obtidos resultados de simulação. Estes serviram para os ajustes que foram necessários na rede neural e no algoritmo, e para a apresentação destes e geração de gráficos que demonstram o desempenho do agente no ambiente.

Os resultados foram adequadamente obtidos conforme os objetivos, e o que foi apresentado nas seções anteriores, foram obtidos resultados de treinamento, pontos aleatórios e, pontos fixos.

O ambiente desenvolvido foi capaz de gerar renderizações do robô Delta em seu ambiente, dessa forma foi possível observar o robô executando ações em suas tentativas de resolver o problema em questão.

Acerca de trabalhos futuros, pode-se avaliar a possibilidade da utilização de um simulador que consiga reproduzir a dinâmica de corpos rígidos do mecanismo, esse simulador deve também dar a possibilidade de se ler e atuador nos atuadores do robô, dessa forma, com a dinâmica do robô se poderia aplicar um algoritmo mais complexo como o DDPG e outros que são capazes de lidar com ambiente contínuos.

Outra possibilidade seria desenvolver um robô físico e implementar um sistema de controle para o atuador do robô através da modelagem matemática dele, com o uso do MATLAB se poderia determinar um controlador ótimo ao robô, além disso, se poderia simular esse controle em uma linha de produção real.

REFERÊNCIAS

ALPHASTAR, THE. **AlphaStar: Mastering the real-time strategy game StarCraft II**. Londres: Deepmind, 2019. Disponível em: https://www.deepmind.com/blog/alphastar-mastering-the-real-time-strategy-game-starcraft-ii/. Acesso em: 10/03/2022.

BONEV, I. **Delta parallel robot – The story of success**. 2001. Disponível em: https://www.parallemic.org/Reviews/Review002.html/. Acesso em: 05/12/2021.

BROWNLEE, J. **What is the Difference Between Test and Validation Datasets?** Machine Learning Mastery, 2017. Disponível em: https://machinelearningmastery.com/difference-test-validation-datasets/. Acesso em: 05/02/2022.

CHI, W. et al. **Design and Experimental Study of a VCM-Based Stewart Parallel Mechanism Used for Active Vibration Isolation**. Energies, v.8, p. 8001-8019.

GÉRON, A. **Hands-On Machine Learning with Scikit-Learn and TensorFlow Concepts, Tools, and Techniques to Build Intelligent Systems**. 1. ed. O'Reilly Media, 2017.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. **Deep Learning**. 1. ed. Cambridge: MIT Press, 2016.

GOUGH V.E.; WHITEHALL S.G. **Universal tyre test machine**. Proceedings of 9th International Congress FISITA, May 1962, p. 117-137.

KERMANY, D.S.; GOLDBAUM, M.; CAI, W. et al. Identifying Medical Diagnoses and Treatable Diseases by Image-Based Deep Learning. **Cell**, v. 172, n. 5, p. 1122-1131, 2018.

KINGMA, D.P.; BA J.L. **Adam: A Method for Stochastic Optimization**. International Conference on Learning Representations (ICLR), 2015.

LOCKHEEDMARTIN. **From the Classroom to the Battlefield: How Lockheed Martin is Advancing AI Technology through University Partnerships**. 2020. Disponível em: https://www.lockheedmartin.com/en-us/news/features/2020/From-the-Classroom-to-the-Battlefield-How-Lockheed-Martin-is-Advancing-AI-Technology.html. Acesso em: 14/05/2022.

MITCHELL, T.M. **Machine Learning**. 1. Ed. McGraw-Hill, 1997.

MNIH, V.; KAVUKCUOGLU, K.; SILVER, D. et al. Human-level control through deep reinforcement learning. **Nature**, v. 518, p. 529–533, 2015.

MORALES, M. **Grokking Deep Reinforcement Learning**. 1 ed. Manning Publications, 2020.

NAIR, A. **Massively Parallel Methods for Deep Reinforcement Learning**. Disponível em: https://arxiv.org/abs/1507.04296. Acesso em 14/11/2022.

PYTORCH CONTRIBUTORS. **RELU**. 2022. Disponível em: https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html. Acesso em: 01/10/2022.
SCHAUL, T. et al. **Prioritized Experience Replay**. International Conference on Learning Representations (ICLR), 2016.

SCIKIT-LEARN. **Underfitting vs. Overfitting**. 20--. Disponível em: https://scikit-learn.org/stable/auto_examples/model_selection/plot_underfitting_overfitting.html. Acesso em: 14/05/2022.

SILVER, D.; HUANG, A.; MADDISON, C.J. et al. Mastering the game of Go with deep neural networks and tree search. **Nature**, v. 529, p. 484–489, 2016.

STAMPER, R. **A Three Degree of Freedom Parallel Manipulator with Only Translational Degrees of Freedom**. 1997.

STEWART. D. **A Platform with Six Degrees of Freedom**. Proceedings of the Institution of Mechanical Engineers, v. 180, p. 371-386, 1965-1966.

SUTTON, R.S.; BARTO, A.G. **Reinforcement Learning: An Introduction**. 2. ed. Cambridge: MIT Press, 2018.

TAGHIRAD, H.D. **Parallel Robots: Mechanics and Control**. 1. ed. CRC Press, 2013.

TSAI, L.W. **Robot Analysis: The Mechanics of Serial and Parallel Manipulators**. 1. ed. Wiley & Sons, 1999.

WILLIAMS, R.L. **The delta parallel robot: kinematics solutions**. 2016. Disponível em: www.ohio.edu/people/williar4/html/pdf/DeltaKin.pdf. Acesso em: 15/05/2022.

YESHMUKHAMETOV, A. et al. **Design and Kinematics of Serial/Parallel Hybrid Robot**. 3rd International Conference on Control, Automation and Robotics (ICCAR). IEEE, 2017.

ZAVATSKY, M. **Delta Robot Kinematics**. 2009. Disponível em: https://hypertriangle.com/~alex/delta-robot-tutorial/. Acesso em 01/10/2022.

ANEXO A – VÍDEO DO ROBÔ

- O vídeo do robô no ambiente pode ser observado no seguinte link do YouTube: https://youtu.be/tuILyU6vk3Y

ANEXO B – CÓDIGO PARA TREINAMENTO

```
    # -*- coding: utf-8 -*-
    """TCC_JULHO_DE_2022_-_TRAINING

    Automatically generated by Colaboratory.

    Original file is located at
        https://colab.research.google.com/drive/1ra7YLzR_XoWpyXK4uKFslGKDwrzkIo0-
    """

    from google.colab import drive
    drive.mount('/drive/')

    import datetime
    import os

    # 0.1*Power[0.999,x]=0.01
    DAY = str(datetime.date.today())
    LEARNING_RATE = 0.0001
    EPS = 0.1
    EPS_DECAY = 0.999
    EPS_MIN = 0.01
    NUMBER_OF_EPISODES = 2302
    EPISODE_TO_START_PRINTING = NUMBER_OF_EPISODES - 10
    DRIVE_PATH = "/drive/MyDrive/TCC_JULHO_DE_2022/TRAININGS"
    RUN_PARAMETERS                                                                  =
f"_LR_=_{LEARNING_RATE}_EPS_DECAY_=_{EPS_DECAY}_EPS_MIN_=_{EPS_MIN}_N_EPISODES_=_{NUMBER_OF_
EPISODES}"
    WEIGHTS_DIR = DRIVE_PATH + "/WEIGHTS/WEIGHTS_DIR_" + DAY + RUN_PARAMETERS
    RENDERS_DIR =  DRIVE_PATH + "/RENDERS/RENDERS_DIR_"  + DAY + RUN_PARAMETERS
    LOGS_DIR = DRIVE_PATH + "/LOGS/LOGS_DIR_" + DAY + RUN_PARAMETERS
    REWARD_DATA_DIR = DRIVE_PATH + "/REWARDS/REWARD_DATA_DIR_" + DAY + RUN_PARAMETERS
    RESULTS_FIG_NAME = DRIVE_PATH + "/RESULTS/RESULTS_FIG_"  + DAY + RUN_PARAMETERS

    print(WEIGHTS_DIR)

    import os

    dir_list   =   [DRIVE_PATH,   WEIGHTS_DIR,   RENDERS_DIR,   LOGS_DIR,   REWARD_DATA_DIR,
RESULTS_FIG_NAME]

    # Check whether the specified path exists or not
    for dir in dir_list:
        isExist = os.path.exists(dir)

        if not isExist:

            # Create a new directory because it does not exist
            os.makedirs(dir)
            print("The new directory is created!")

    import os
    import torch as T
    import torch.nn as nn
    import torch.optim as optim
    import torch.nn.functional as F


    class QNetwork(nn.Module):
        def       __init__(self,       lr=LEARNING_RATE,       n_states=4,       n_actions=6,
checkpoint_dir=f"./{WEIGHTS_DIR}", filename=f"{WEIGHTS_DIR}"):
            super(QNetwork, self).__init__()
            if not os.path.isdir(checkpoint_dir):
                os.makedirs(checkpoint_dir)
            self.checkpoint_file = os.path.join(checkpoint_dir, filename)
            # Detalhe da rede neural
            self.fc1 = nn.Linear(n_states, 300)
            self.fc2 = nn.Linear(300, 400)
            self.fc3 = nn.Linear(400, 600)
```

```
        self.fc4 = nn.Linear(600, 600)
        self.fc5 = nn.Linear(600, 400)
        self.fc6 = nn.Linear(400, 300)
        self.fc7 = nn.Linear(300, n_actions)
        # Optimizer e loss
        self.optimizer = optim.Adam(self.parameters(), lr=lr)
        self.loss = nn.MSELoss()
        self.device = T.device('cuda:0' if T.cuda.is_available() else 'cpu')
        self.to(self.device)

    def forward(self, state):
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        x = F.relu(self.fc4(x))
        x = F.relu(self.fc5(x))
        x = F.relu(self.fc6(x))
        return self.fc7(x)

    def save_checkpoint(self):
        print('... Save checkpoint ...')
        T.save(self.state_dict(), self.checkpoint_file)

    def load_checkpoint(self):
        print('... Load checkpoint ...')
        self.load_state_dict(T.load(self.checkpoint_file))
```

```
import numpy as np


class ReplayMemory(object):
    def __init__(self, max_size=10000, n_states=4):
        self.max_size = max_size
        self.memory_counter = 0
        self.states_memory = np.zeros((max_size, n_states), dtype=np.float32)
        self.next_states_memory = np.zeros(
            (max_size, n_states), dtype=np.float32)
        self.actions_memory = np.zeros(max_size, dtype=np.int64)
        self.rewards_memory = np.zeros(max_size, dtype=np.float32)
        self.dones_memory = np.zeros(max_size, dtype=bool)

    def store_transition(self, state, action, reward, next_state, done):
        index = self.memory_counter % self.max_size
        self.states_memory[index] = state
        self.actions_memory[index] = action
        self.rewards_memory[index] = reward
        self.next_states_memory[index] = next_state
        self.dones_memory[index] = done
        self.memory_counter += 1

    def sample_memory(self, batch_size):
        max_mem = min(self.memory_counter, self.max_size)
        batch = np.random.choice(max_mem, batch_size, replace=False)
        states = self.states_memory[batch]
        actions = self.actions_memory[batch]
        rewards = self.rewards_memory[batch]
        next_states = self.next_states_memory[batch]
        dones = self.dones_memory[batch]
        return states, actions, rewards, next_states, dones
```

```
import numpy as np
import torch as T

class DQNAgent(object):
    def __init__(
        self,
        alpha=0.0005,
        gamma=0.99,
        eps=EPS,
        eps_decay=EPS_DECAY,
        eps_min=EPS_MIN,
```

```
            tau=0.001,
            max_size=100000,
            batch_size=64,
            update_rate=4,
            n_states=0,
            n_actions=0,
            checkpoint_dir=f"{WEIGHTS_DIR}",
        ):

            self.gamma = gamma
            self.eps = eps
            self.eps_decay = eps_decay
            self.eps_min = eps_min
            self.tau = tau
            self.batch_size = batch_size
            self.update_rate = update_rate
            self.action_space = [i for i in range(n_actions)]

            # Replay memory
            self.memory = ReplayMemory(max_size=max_size, n_states=n_states)

            # Q-Network
            self.qnetwork_local = QNetwork(lr=alpha, n_states=n_states, n_actions=n_actions,
                                           checkpoint_dir=checkpoint_dir,
filename="qnetwork_local_TRAININGS.pth")
            self.qnetwork_target = QNetwork(lr=alpha, n_states=n_states, n_actions=n_actions,
                                            checkpoint_dir=checkpoint_dir,
filename="qnetwork_target_TRAININGS.pth")

            self.counter = 0

        def decrement_epsilon(self):
            self.eps *= self.eps_decay
            if self.eps < self.eps_min:
                self.eps = self.eps_min

        def epsilon_greedy(self, state):
            if np.random.random() > self.eps:
                state = T.tensor([state], dtype=T.float).to(
                    self.qnetwork_local.device)
                actions = self.qnetwork_local.forward(state)
                action = T.argmax(actions).item()
            else:
                action = np.random.choice(self.action_space)
            return action

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

        def sample_memory(self):
            states, actions, rewards, next_states, dones = \
                self.memory.sample_memory(self.batch_size)
            t_states = T.tensor(states).to(self.qnetwork_local.device)
            t_actions = T.tensor(actions).to(self.qnetwork_local.device)
            t_rewards = T.tensor(rewards).to(self.qnetwork_local.device)
            t_next_states = T.tensor(next_states).to(self.qnetwork_local.device)
            t_dones = T.tensor(dones).to(self.qnetwork_local.device)
            return t_states, t_actions, t_rewards, t_next_states, t_dones

        def save_models(self):
            self.qnetwork_local.save_checkpoint()
            self.qnetwork_target.save_checkpoint()

        def load_models(self):
            self.qnetwork_local.load_checkpoint()
            self.qnetwork_target.load_checkpoint()

        def learn(self, state, action, reward, next_state, done):
            # Save experience to memory
            self.store_transition(state, action, reward, next_state, done)
```

```
            # If not enough memory then skip learning
            if self.memory.memory_counter < self.batch_size:
                return

            # Update target network parameter every update rate
            if self.counter % self.update_rate == 0:
                self.soft_update(self.tau)

            # Take random sampling from memory
            states, actions, rewards, next_states, dones = self.sample_memory()

            # Update action value
            indices = np.arange(self.batch_size)
            q_pred = self.qnetwork_local.forward(states)[indices, actions]
            q_next = self.qnetwork_target.forward(next_states).max(dim=1)[0]
            q_next[dones] = 0.0
            q_target = rewards + self.gamma*q_next
            self.qnetwork_local.optimizer.zero_grad()
            loss = \
                self.qnetwork_local.loss(q_target, q_pred) \
                .to(self.qnetwork_local.device)
            loss.backward()
            self.qnetwork_local.optimizer.step()
            self.counter += 1

        def soft_update(self, tau):
            for   target_param,   local_param   in   zip(self.qnetwork_target.parameters(),
self.qnetwork_local.parameters()):
                target_param.data.copy_(
                    tau*local_param.data + (1.0-tau)*target_param.data)

        def regular_update(self):
            for   target_param,   local_param   in   zip(self.qnetwork_target.parameters(),
self.qnetwork_local.parameters()):
                target_param.data.copy_(local_param.data + target_param.data)
```

```
    from mpl_toolkits.mplot3d import Axes3D
    import os
    import random
    import numpy as np
    import time
    from gym import spaces
    import math as mt
    import matplotlib.pyplot as plt
    import matplotlib
    matplotlib.use('Agg')


    class DeltaEnv():
        def __init__(self):

            self.min_theta = -1.1
            self.max_theta = 1.1
            self.theta = np.array([0.0, 0.0, 0.0])

            self.distance = 100
            self.current_distance = 100

            self.set_increment_rate(0.1)
            self.render_counter = 0

            self.action = {
                0: "HOLD",
                1: "INC_J1",
                2: "DEC_J1",
                3: "INC_J2",
                4: "DEC_J2",
                5: "INC_J3",
                6: "DEC_J3",
                7: "INC_J1_J2",
                8: "DEC_J1_J2",
```

```
            9: "INC_J2_J3",
            10: "DEC_J2_J3",
            11: "INC_J1_J3",
            12: "DEC_J1_J3",
            13: "INC_J1_J2_J3",
            14: "DEC_J1_J2_J3"}

        self.ee_pos = self.forward_kinematics(
            self.theta[0], self.theta[1], self.theta[2])[0]
        self.goal_pos = self.generate_random_positions()[1]
        self.states = np.hstack(
            (self.goal_pos, self.ee_pos, self.theta, self.distance, 0, 0))

        self.action_space = spaces.Discrete(len(self.action))
        self.observation_space = spaces.Discrete(len(self.states))

    def forward_kinematics(self, t1, t2, t3):
        '''
            Esta função recebe como entrada angulos: theta_1, theta_2, theta_3 e
            retorna o ponto no espaço no qual o efetuador do robô deve se
            posicionar.
        '''
        X, Y, Z = 0, 0, 0

        L = 400  # mm
        l = 900
        rA = 180
        rE = 100

        # t1 = np.deg2rad(t1)
        # t2 = np.deg2rad(t2)
        # t3 = np.deg2rad(t3)

        phi = np.deg2rad(30)
        r = rA - rE

        x1 = 0
        y1 = - (r + L * mt.cos(t1))
        z1 = - L * mt.sin(t1)
        ponto_1 = np.array((x1, y1, z1))

        x2 = (r + L * mt.cos(t2)) * mt.cos(phi)
        y2 = (r + L * mt.cos(t2)) * mt.sin(phi)
        z2 = - L * mt.sin(t2)
        ponto_2 = np.array((x2, y2, z2))

        x3 = - (r + L * mt.cos(t3)) * mt.cos(phi)
        y3 = (r + L * mt.cos(t3)) * mt.sin(phi)
        z3 = - L * mt.sin(t3)
        ponto_3 = np.array((x3, y3, z3))

        p1 = y1**2 + z1**2
        p2 = x2**2 + y2**2 + z2**2
        p3 = x3**2 + y3**2 + z3**2

        a1 = (z2 - z1) * (y3 - y1) - (z3 - z1) * (y2 - y1)
        b1 = - ((p2 - p1) * (y3 - y1) - (p3 - p1) * (y2 - y1)) / 2

        a2 = - (z2 - z1) * x3 + (z3 - z1) * x2
        b2 = ((p2 - p1) * x3 - (p3 - p1) * x2) / 2

        dnm = (y2 - y1) * x3 - (y3 - y1) * x2

        a = a1**2 + a2**2 + dnm**2
        b = 2 * (a1 * b1 + a2 * (b2 - y1 * dnm) - z1 * dnm**2)
        c = (b2 - y1 * dnm) * (b2 - y1 * dnm) + \
            b1**2 + (dnm**2) * (z1**2 - l**2)

        d = b * b - 4.0 * a * c

        if (d < 0):
```

```
            Z = -1
            b + mt.sqrt(-d)
            b
            a
        else:
            Z = - 0.5 * (b + mt.sqrt(d)) / a
            X = (a1 * Z + b1) / dnm
            Y = (a2 * Z + b2) / dnm

        ee_pos = np.array((X, Y, Z))

        return ee_pos, ponto_1, ponto_2, ponto_3

    def set_increment_rate(self, rate):
        self.rate = rate

    def step(self, action):
        if self.action[action] == "HOLD":
            self.theta[0] += 0  # self.rate
            self.theta[1] += 0  # self.rate
            self.theta[2] += 0  # self.rate
        elif self.action[action] == "INC_J1":
            self.theta[0] += self.rate
        elif self.action[action] == "DEC_J1":
            self.theta[0] -= self.rate
        elif self.action[action] == "INC_J2":
            self.theta[1] += self.rate
        elif self.action[action] == "DEC_J2":
            self.theta[1] -= self.rate
        elif self.action[action] == "INC_J3":
            self.theta[2] += self.rate
        elif self.action[action] == "DEC_J3":
            self.theta[2] -= self.rate
        elif self.action[action] == "INC_J1_J2":
            self.theta[0] += self.rate
            self.theta[1] += self.rate
        elif self.action[action] == "DEC_J1_J2":
            self.theta[0] -= self.rate
            self.theta[1] -= self.rate
        elif self.action[action] == "INC_J2_J3":
            self.theta[1] += self.rate
            self.theta[2] += self.rate
        elif self.action[action] == "DEC_J2_J3":
            self.theta[1] -= self.rate
            self.theta[2] -= self.rate
        elif self.action[action] == "INC_J1_J3":
            self.theta[0] += self.rate
            self.theta[2] += self.rate
        elif self.action[action] == "DEC_J1_J3":
            self.theta[0] -= self.rate
            self.theta[2] -= self.rate
        elif self.action[action] == "INC_J1_J2_J3":
            self.theta[0] += self.rate
            self.theta[1] += self.rate
            self.theta[2] += self.rate
        elif self.action[action] == "DEC_J1_J2_J3":
            self.theta[0] -= self.rate
            self.theta[1] -= self.rate
            self.theta[2] -= self.rate

        self.ee_pos = self.forward_kinematics(
            self.theta[0], self.theta[1], self.theta[2])[0]

        self.theta[0] = np.clip(self.theta[0], self.min_theta, self.max_theta)
        self.theta[1] = np.clip(self.theta[1], self.min_theta, self.max_theta)
        self.theta[2] = np.clip(self.theta[2], self.min_theta, self.max_theta)

        self.theta[0] = self.normalize_angle(self.theta[0])
        self.theta[1] = self.normalize_angle(self.theta[1])
        self.theta[2] = self.normalize_angle(self.theta[2])
```

```
        a = np.array((self.goal[0], self.goal[1], self.goal[2]))
        b = np.array((self.ee_pos[0], self.ee_pos[1], self.ee_pos[2]))

        self.distance = np.linalg.norm(a-b)
        #print("DISTANCIA DELTA ENV: ", distance)

        done = False
        reward = 0
        if self.distance >= self.current_distance:
            reward = -1

        epsilon = 50

        if (self.distance > -epsilon and self.distance < epsilon):
            reward = 1
            done = True

        self.current_distance = self.distance
        self.current_score += reward

        if self.current_score == -10 or self.current_score >= 10:
            done = True
        else:
            done = False

        observation = np.hstack(
            (self.goal_pos, self.ee_pos, self.theta, self.distance))
        info = {
            'distance': self.distance,
            'goal_position': self.goal_pos,
            'ee_position': self.ee_pos
        }
        #self.render(self.theta, self.goal_pos)
        # self.timerate.sleep()
        return observation, reward, done, info

    def generate_random_positions(self):
        angles = np.arange(-1.1, 1.1, 0.01).tolist()

        theta0 = random.choice(angles)
        theta1 = random.choice(angles)
        theta2 = random.choice(angles)

        self.goal = self.forward_kinematics(theta0, theta1, theta2)[0]

        return self.goal

    @staticmethod
    def normalize_angle(angle):
        return mt.atan2(mt.sin(angle), mt.cos(angle))

    def reset(self):
        self.goal_pos = self.generate_random_positions()

        self.theta[0] = 0
        self.theta[1] = 0
        self.theta[2] = 0

        self.current_score = 0

        self.ee_pos = self.forward_kinematics(
            self.theta[0], self.theta[1], self.theta[2])[0]
        observation = np.hstack(
            (self.goal_pos, self.ee_pos, self.theta, self.distance))

        return observation

    def render(self, theta, goal_pos):
        data = self.forward_kinematics(theta[0], theta[1], theta[2])

        ee_pos = data[0]
```

```
            ponto_1 = data[1]
            ponto_2 = data[2]
            ponto_3 = data[3]
            fig = plt.figure(1, figsize=(5, 5))

            ax = fig.add_subplot(111, projection='3d')

            # plot the point (2,3,4) on the figure
            ax.scatter(ee_pos[0], ee_pos[1], ee_pos[2], c='black', marker='o')
            # plot the point (2,3,4) on the figure
            ax.scatter(ponto_1[0], ponto_1[1], ponto_1[2], c='blue', marker='$O$')
            # plot the point (2,3,4) on the figure
            ax.scatter(ponto_2[0], ponto_2[1], ponto_2[2], c='blue', marker='$O$')
            # plot the point (2,3,4) on the figure
            ax.scatter(ponto_3[0], ponto_3[1], ponto_3[2], c='blue', marker='$O$')
            ax.scatter(goal_pos[0], goal_pos[1],
                       goal_pos[2], c='red', marker='$X$')

            t = np.linspace(0, 2*np.pi, 1000)
            z_line = ee_pos[2]  # np.linspace(0, 15, 1000)
            x_line = ee_pos[0] + 100*np.cos(t)
            y_line = ee_pos[1] + 100*np.sin(t)

            z_line1 = 0  # np.linspace(0, 15, 1000)
            x_line1 = 180*np.cos(t)
            y_line1 = 180*np.sin(t)

            ax.plot3D(x_line, y_line, z_line, 'blue')
            ax.plot3D(x_line1, y_line1, z_line1, 'red')

            x = [0,    ponto_1[0], 180 *
                 np.sin(np.deg2rad(60)), ponto_2[0], -180*np.sin(np.deg2rad(60)), ponto_3[0]]
            y = [-180,    ponto_1[1], 180 *
                 np.cos(np.deg2rad(60)), ponto_2[1],  180*np.cos(np.deg2rad(60)), ponto_3[1]]
            z = [0,    ponto_1[2], 0, ponto_2[2],
                 0,                        ponto_3[2]]

            x1 = [0 + ee_pos[0],    ponto_1[0], 100*np.sin(np.deg2rad(
                60))  +  ee_pos[0],  ponto_2[0],  -100*np.sin(np.deg2rad(60))  +  ee_pos[0],
ponto_3[0]]
            y1 = [-100 + ee_pos[1],    ponto_1[1], 100*np.cos(np.deg2rad(
                60))  +  ee_pos[1],  ponto_2[1],    100*np.cos(np.deg2rad(60))  +  ee_pos[1],
ponto_3[1]]
            z1 = [ee_pos[2],    ponto_1[2],                                ee_pos[2],
                  ponto_2[2],                                 ee_pos[2], ponto_3[2]]

            plt.plot(x, y, z,  'ro')

            def connectpoints(x, y, z, p1, p2):
                x1, x2 = x[p1], x[p2]
                y1, y2 = y[p1], y[p2]
                z1, z2 = z[p1], z[p2]
                plt.plot([x1, x2], [y1, y2], [z1, z2], 'k-')

            connectpoints(x, y, z, 0, 1)
            connectpoints(x, y, z, 2, 3)
            connectpoints(x, y, z, 4, 5)

            connectpoints(x1, y1, z1, 0, 1)
            connectpoints(x1, y1, z1, 2, 3)
            connectpoints(x1, y1, z1, 4, 5)

            ax.view_init(elev=30, azim=60)

            ax.axes.set_xlim3d(left=-650, right=650)
            ax.axes.set_ylim3d(bottom=-650, top=650)
            ax.axes.set_zlim3d(bottom=-1200, top=100)

            self.render_counter += 1
            plt.savefig(
                f"{RENDERS_DIR}/step_{self.render_counter}")
```

```
            ax.cla()
            plt.close(fig)

    from matplotlib import pyplot as plt
    import numpy as np
    from torch.utils.tensorboard import SummaryWriter

    if __name__ == "__main__":

        env = DeltaEnv()
        n_states = env.observation_space.n
        n_actions = env.action_space.n
        agent = DQNAgent(alpha=0.0001, n_states=n_states, n_actions=n_actions)
        writer = SummaryWriter(f'./log/{LOGS_DIR}')
        load_models = False
        n_episodes = NUMBER_OF_EPISODES
        n_steps = 500

        # Load weights
        if load_models:
            agent.eps = agent.eps_min
            agent.load_models()

        total_reward_hist = []
        avg_reward_hist = []
        for episode in range(1, n_episodes + 1):
            state = env.reset()
            total_reward = 0
            for t in range(n_steps):
                # Render after episode 1800
                if episode > EPISODE_TO_START_PRINTING:
                    env.render(env.theta, env.goal_pos)
                action = agent.epsilon_greedy(state)
                next_state, reward, done, info = env.step(action)
                # print(info)
                agent.learn(state, action, reward, next_state, done)
                state = next_state
                total_reward += reward
                if done:
                    break
            if not load_models:
                agent.decrement_epsilon()

            # Save model
            if episode > 100 and episode % 20 == 0:
                agent.save_models()

            # env.render()

            total_reward_hist.append(total_reward)
            avg_reward = np.average(total_reward_hist[-100:])
            avg_reward_hist.append(avg_reward)
            print("Episode :", episode, "Epsilon : {:.4f}".format(agent.eps), "Total Reward :
{:.4f}".format(
                total_reward), "Avg Reward : {:.4f}".format(avg_reward))
            with open(f"{REWARD_DATA_DIR}_REWARD_DATA.txt", "a") as myfile1:
                myfile1.write("%.4f, %.4f, %.4f, %.4f\n" % (episode, agent.eps, total_reward,
avg_reward))
            # Tensorboard log
            writer.add_scalar('Total Reward', total_reward, episode)
            writer.add_scalar('Avg Reward', avg_reward, episode)
            # writer.add_scalar('Epsilon', agent.eps, episode)
        fig, ax = plt.subplots(1)
        t = np.arange(n_episodes)
        ax.plot(t, total_reward_hist, label="Recompensas Totais")
        ax.plot(t, avg_reward_hist, label="Recompensa média")
        ax.set_title("Recompensas vs Episódios")
        ax.set_xlabel("Episódio")
        ax.set_ylabel("Recompensa")
        ax.legend()
        plt.savefig(f"{RESULTS_FIG_NAME}.png")
```

```
        ax.cla()
        plt.close(fig)

    # import pandas as pd

    #     file_path     =     DRIVE_PATH     +     "/REWARDS/"+     "REWARD_DATA_DIR_2022-07-
24_LR_=_0.0001_EPS_DECAY_=_0.999_EPS_MIN_=_0.001_N_EPISODES_=_6904_REWARD_DATA.txt"
    #         print(DRIVE_PATH         +         "/REWARDS/"+         "REWARD_DATA_DIR_2022-07-
31_LR_=_0.0001_EPS_DECAY_=_0.999_EPS_MIN_=_0.001_N_EPISODES_=_6904_REWARD_DATA")


    # file1 = pd.read_csv(file_path)
    # file1
    # # count = 0

    # numpyfile1 = file1.to_numpy()
    # print(numpyfile1)
    # for reward in numpyfile1:
    #     print(reward)
    #     if(reward[2] == 10):
    #         count+=1
    # count

    import pandas as pd
    DRIVE_PATH = "/drive/MyDrive/TCC_JULHO_DE_2022/TESTS"

    file_path      =      DRIVE_PATH      +      "/REWARDS/"+      "REWARD_DATA_DIR_2022-08-
20_LR_=_0.0001_EPS_DECAY_=_0.999_EPS_MIN_=_0.01_N_EPISODES_=_2302_REWARD_DATA.txt"
    print(DRIVE_PATH           +           "/REWARDS/"+           "REWARD_DATA_DIR_2022-08-
01_LR_=_0.0001_EPS_DECAY_=_0.999_EPS_MIN_=_0.001_N_EPISODES_=_6904_REWARD_DATA")


    file1 = pd.read_csv(file_path)
    file1.to_numpy
    count = 0

    numpyfile1 = file1.to_numpy()
    print(numpyfile1)
    for reward in numpyfile1:
        # print(reward)
        if(reward[2] == 10):
            count+=1
    count
```

ANEXO C – CÓDIGO PARA TESTES COM PONTOS ALEATÓRIOS

```
    # -*- coding: utf-8 -*-
    """TCC_JULHO_DE_2022_-_TESTS

    Automatically generated by Colaboratory.

    Original file is located at
        https://colab.research.google.com/drive/1bLwWzSy8VdNWmfYlRG5IuonUEuGaxQg9
    """

    from google.colab import drive
    drive.mount('/drive/')

    import datetime
    import os

    # 0.1*Power[0.999,x]=0.01
    DAY = str(datetime.date.today())
    LEARNING_RATE = 0.0001
    EPS = 0.1
    EPS_DECAY = 0.999
    EPS_MIN = 0.01
    NUMBER_OF_EPISODES = 10
    EPISODE_TO_START_PRINTING = # NUMBER_OF_EPISODES - 10
    DRIVE_PATH = "/drive/MyDrive/TCC_JULHO_DE_2022/TESTS"
    RUN_PARAMETERS                                                                  =
f"_LR_=_{LEARNING_RATE}_EPS_DECAY_=_{EPS_DECAY}_EPS_MIN_=_{EPS_MIN}_N_EPISODES_=_{NUMBER_OF_
EPISODES}"
    WEIGHTS_DIR  =  "/drive/MyDrive/TCC_JULHO_DE_2022/TRAININGS/WEIGHTS/WEIGHTS_DIR_2022-08-
20_LR_=_0.0001_EPS_DECAY_=_0.999_EPS_MIN_=_0.01_N_EPISODES_=_2302"     #     DRIVE_PATH     +
"/WEIGHTS/WEIGHTS_DIR_" + DAY + RUN_PARAMETERS
    RENDERS_DIR =  DRIVE_PATH + "/RENDERS/RENDERS_DIR_"  + DAY + RUN_PARAMETERS
    LOGS_DIR = DRIVE_PATH + "/LOGS/LOGS_DIR_" + DAY + RUN_PARAMETERS
    REWARD_DATA_DIR = DRIVE_PATH + "/REWARDS/REWARD_DATA_DIR_" + DAY + RUN_PARAMETERS
    RESULTS_FIG_NAME = DRIVE_PATH + "/RESULTS/RESULTS_FIG_"  + DAY + RUN_PARAMETERS

    WEIGHTS_DIR

    import os

    dir_list   =   [DRIVE_PATH,   WEIGHTS_DIR,   RENDERS_DIR,   LOGS_DIR,   REWARD_DATA_DIR,
RESULTS_FIG_NAME]

    # Check whether the specified path exists or not
    for dir in dir_list:
        isExist = os.path.exists(dir)

        if not isExist:

            # Create a new directory because it does not exist
            os.makedirs(dir)
            print("The new directory is created!")

    import os
    import torch as T
    import torch.nn as nn
    import torch.optim as optim
    import torch.nn.functional as F


    class QNetwork(nn.Module):
        def       __init__(self,       lr=LEARNING_RATE,       n_states=4,       n_actions=6,
checkpoint_dir=f"./{WEIGHTS_DIR}", filename=f"{WEIGHTS_DIR}"):
            super(QNetwork, self).__init__()
            if not os.path.isdir(checkpoint_dir):
                os.makedirs(checkpoint_dir)
            self.checkpoint_file = os.path.join(checkpoint_dir, filename)
            # Detalhe da rede neural
            self.fc1 = nn.Linear(n_states, 300)
```

```
        self.fc2 = nn.Linear(300, 400)
        self.fc3 = nn.Linear(400, 600)
        self.fc4 = nn.Linear(600, 600)
        self.fc5 = nn.Linear(600, 400)
        self.fc6 = nn.Linear(400, 300)
        self.fc7 = nn.Linear(300, n_actions)
        # Optimizer e loss
        self.optimizer = optim.Adam(self.parameters(), lr=lr)
        self.loss = nn.MSELoss()
        self.device = T.device('cuda:0' if T.cuda.is_available() else 'cpu')
        self.to(self.device)

    def forward(self, state):
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        x = F.relu(self.fc4(x))
        x = F.relu(self.fc5(x))
        x = F.relu(self.fc6(x))
        return self.fc7(x)

    def save_checkpoint(self):
        print('... Save checkpoint ...')
        T.save(self.state_dict(), self.checkpoint_file)

    def load_checkpoint(self):
        print('... Load checkpoint ...')
        self.load_state_dict(T.load(self.checkpoint_file))

import numpy as np


class ReplayMemory(object):
    def __init__(self, max_size=10000, n_states=4):
        self.max_size = max_size
        self.memory_counter = 0
        self.states_memory = np.zeros((max_size, n_states), dtype=np.float32)
        self.next_states_memory = np.zeros(
            (max_size, n_states), dtype=np.float32)
        self.actions_memory = np.zeros(max_size, dtype=np.int64)
        self.rewards_memory = np.zeros(max_size, dtype=np.float32)
        self.dones_memory = np.zeros(max_size, dtype=bool)

    def store_transition(self, state, action, reward, next_state, done):
        index = self.memory_counter % self.max_size
        self.states_memory[index] = state
        self.actions_memory[index] = action
        self.rewards_memory[index] = reward
        self.next_states_memory[index] = next_state
        self.dones_memory[index] = done
        self.memory_counter += 1

    def sample_memory(self, batch_size):
        max_mem = min(self.memory_counter, self.max_size)
        batch = np.random.choice(max_mem, batch_size, replace=False)
        states = self.states_memory[batch]
        actions = self.actions_memory[batch]
        rewards = self.rewards_memory[batch]
        next_states = self.next_states_memory[batch]
        dones = self.dones_memory[batch]
        return states, actions, rewards, next_states, dones

np.zeros(10000, dtype=np.int64)

import numpy as np
import torch as T

class DQNAgent(object):
    def __init__(
        self,
        alpha=0.0005,
```

```
            gamma=0.99,
            eps=EPS,
            eps_decay=EPS_DECAY,
            eps_min=EPS_MIN,
            tau=0.001,
            max_size=100000,
            batch_size=64,
            update_rate=4,
            n_states=0,
            n_actions=0,
            checkpoint_dir=f"{WEIGHTS_DIR}",
        ):

            self.gamma = gamma
            self.eps = eps
            self.eps_decay = eps_decay
            self.eps_min = eps_min
            self.tau = tau
            self.batch_size = batch_size
            self.update_rate = update_rate
            self.action_space = [i for i in range(n_actions)]

            # Replay memory
            self.memory = ReplayMemory(max_size=max_size, n_states=n_states)

            # Q-Network
            self.qnetwork_local = QNetwork(lr=alpha, n_states=n_states, n_actions=n_actions,
                                           checkpoint_dir=checkpoint_dir,
filename="qnetwork_local_TESTE.pth")
            self.qnetwork_target = QNetwork(lr=alpha, n_states=n_states, n_actions=n_actions,
                                            checkpoint_dir=checkpoint_dir,
filename="qnetwork_target_TESTE.pth")

            self.counter = 0

        def decrement_epsilon(self):
            self.eps *= self.eps_decay
            if self.eps < self.eps_min:
                self.eps = self.eps_min

        def epsilon_greedy(self, state):
            if np.random.random() > self.eps:
                state = T.tensor([state], dtype=T.float).to(
                    self.qnetwork_local.device)
                actions = self.qnetwork_local.forward(state)
                action = T.argmax(actions).item()
            else:
                action = np.random.choice(self.action_space)
            return action

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

        def sample_memory(self):
            states, actions, rewards, next_states, dones = \
                self.memory.sample_memory(self.batch_size)
            t_states = T.tensor(states).to(self.qnetwork_local.device)
            t_actions = T.tensor(actions).to(self.qnetwork_local.device)
            t_rewards = T.tensor(rewards).to(self.qnetwork_local.device)
            t_next_states = T.tensor(next_states).to(self.qnetwork_local.device)
            t_dones = T.tensor(dones).to(self.qnetwork_local.device)
            return t_states, t_actions, t_rewards, t_next_states, t_dones

        def save_models(self):
            self.qnetwork_local.save_checkpoint()
            self.qnetwork_target.save_checkpoint()

        def load_models(self):
            self.qnetwork_local.load_checkpoint()
            self.qnetwork_target.load_checkpoint()
```

```
        def learn(self, state, action, reward, next_state, done):
            # Save experience to memory
            self.store_transition(state, action, reward, next_state, done)

            # If not enough memory then skip learning
            if self.memory.memory_counter < self.batch_size:
                return

            # Update target network parameter every update rate
            if self.counter % self.update_rate == 0:
                self.soft_update(self.tau)

            # Take random sampling from memory
            states, actions, rewards, next_states, dones = self.sample_memory()

            # Update action value
            indices = np.arange(self.batch_size)
            q_pred = self.qnetwork_local.forward(states)[indices, actions]
            q_next = self.qnetwork_target.forward(next_states).max(dim=1)[0]
            q_next[dones] = 0.0
            q_target = rewards + self.gamma*q_next
            self.qnetwork_local.optimizer.zero_grad()
            loss = \
                self.qnetwork_local.loss(q_target, q_pred) \
                .to(self.qnetwork_local.device)
            loss.backward()
            self.qnetwork_local.optimizer.step()
            self.counter += 1

        def soft_update(self, tau):
            for   target_param,   local_param   in   zip(self.qnetwork_target.parameters(),
self.qnetwork_local.parameters()):
                target_param.data.copy_(
                    tau*local_param.data + (1.0-tau)*target_param.data)

        def regular_update(self):
            for   target_param,   local_param   in   zip(self.qnetwork_target.parameters(),
self.qnetwork_local.parameters()):
                target_param.data.copy_(local_param.data + target_param.data)

    from mpl_toolkits.mplot3d import Axes3D
    import os
    import random
    import numpy as np
    import time
    from gym import spaces
    import math as mt
    import matplotlib.pyplot as plt
    import matplotlib
    matplotlib.use('Agg')


    class DeltaEnv():
        def __init__(self):

            self.min_theta = -1.1
            self.max_theta = 1.1
            self.theta = np.array([0.0, 0.0, 0.0])

            self.distance = 1000
            self.current_distance = 1000

            self.set_increment_rate(0.1)
            self.render_counter = 0

            self.action = {
                0: "HOLD",
                1: "INC_J1",
                2: "DEC_J1",
                3: "INC_J2",
                4: "DEC_J2",
```

```
            5: "INC_J3",
            6: "DEC_J3",
            7: "INC_J1_J2",
            8: "DEC_J1_J2",
            9: "INC_J2_J3",
            10: "DEC_J2_J3",
            11: "INC_J1_J3",
            12: "DEC_J1_J3",
            13: "INC_J1_J2_J3",
            14: "DEC_J1_J2_J3"}

        self.ee_pos = self.forward_kinematics(
            self.theta[0], self.theta[1], self.theta[2])[0]
        self.goal_pos = self.generate_random_positions()[0]
        self.states = np.hstack(
            (self.goal_pos, self.ee_pos, self.theta, self.distance, 0, 0))

        self.action_space = spaces.Discrete(len(self.action))
        self.observation_space = spaces.Discrete(len(self.states))

    def forward_kinematics(self, t1, t2, t3):
        '''
            Esta função recebe como entrada angulos: theta_1, theta_2, theta_3 e
            retorna o ponto no espaço no qual o efetuador do robô deve se
            posicionar.
        '''
        X, Y, Z = 0, 0, 0

        L = 400  # mm
        l = 900
        rA = 180
        rE = 100

        # t1 = np.deg2rad(t1)
        # t2 = np.deg2rad(t2)
        # t3 = np.deg2rad(t3)

        phi = np.deg2rad(30)
        r = rA - rE

        x1 = 0
        y1 = - (r + L * mt.cos(t1))
        z1 = - L * mt.sin(t1)
        ponto_1 = np.array((x1, y1, z1))

        x2 = (r + L * mt.cos(t2)) * mt.cos(phi)
        y2 = (r + L * mt.cos(t2)) * mt.sin(phi)
        z2 = - L * mt.sin(t2)
        ponto_2 = np.array((x2, y2, z2))

        x3 = - (r + L * mt.cos(t3)) * mt.cos(phi)
        y3 = (r + L * mt.cos(t3)) * mt.sin(phi)
        z3 = - L * mt.sin(t3)
        ponto_3 = np.array((x3, y3, z3))

        p1 = y1**2 + z1**2
        p2 = x2**2 + y2**2 + z2**2
        p3 = x3**2 + y3**2 + z3**2

        a1 = (z2 - z1) * (y3 - y1) - (z3 - z1) * (y2 - y1)
        b1 = - ((p2 - p1) * (y3 - y1) - (p3 - p1) * (y2 - y1)) / 2

        a2 = - (z2 - z1) * x3 + (z3 - z1) * x2
        b2 = ((p2 - p1) * x3 - (p3 - p1) * x2) / 2

        dnm = (y2 - y1) * x3 - (y3 - y1) * x2

        a = a1**2 + a2**2 + dnm**2
        b = 2 * (a1 * b1 + a2 * (b2 - y1 * dnm) - z1 * dnm**2)
        c = (b2 - y1 * dnm) * (b2 - y1 * dnm) + \
            b1**2 + (dnm**2) * (z1**2 - l**2)
```

```
    d = b * b - 4.0 * a * c

    if (d < 0):
        Z = -1
        b + mt.sqrt(-d)
        b
        a
    else:
        Z = - 0.5 * (b + mt.sqrt(d)) / a
        X = (a1 * Z + b1) / dnm
        Y = (a2 * Z + b2) / dnm

    ee_pos = np.array((X, Y, Z))

    return ee_pos, ponto_1, ponto_2, ponto_3

def set_increment_rate(self, rate):
    self.rate = rate

def step(self, action):
    if self.action[action] == "HOLD":
        self.theta[0] += 0  # self.rate
        self.theta[1] += 0  # self.rate
        self.theta[2] += 0  # self.rate
    elif self.action[action] == "INC_J1":
        self.theta[0] += self.rate
    elif self.action[action] == "DEC_J1":
        self.theta[0] -= self.rate
    elif self.action[action] == "INC_J2":
        self.theta[1] += self.rate
    elif self.action[action] == "DEC_J2":
        self.theta[1] -= self.rate
    elif self.action[action] == "INC_J3":
        self.theta[2] += self.rate
    elif self.action[action] == "DEC_J3":
        self.theta[2] -= self.rate
    elif self.action[action] == "INC_J1_J2":
        self.theta[0] += self.rate
        self.theta[1] += self.rate
    elif self.action[action] == "DEC_J1_J2":
        self.theta[0] -= self.rate
        self.theta[1] -= self.rate
    elif self.action[action] == "INC_J2_J3":
        self.theta[1] += self.rate
        self.theta[2] += self.rate
    elif self.action[action] == "DEC_J2_J3":
        self.theta[1] -= self.rate
        self.theta[2] -= self.rate
    elif self.action[action] == "INC_J1_J3":
        self.theta[0] += self.rate
        self.theta[2] += self.rate
    elif self.action[action] == "DEC_J1_J3":
        self.theta[0] -= self.rate
        self.theta[2] -= self.rate
    elif self.action[action] == "INC_J1_J2_J3":
        self.theta[0] += self.rate
        self.theta[1] += self.rate
        self.theta[2] += self.rate
    elif self.action[action] == "DEC_J1_J2_J3":
        self.theta[0] -= self.rate
        self.theta[1] -= self.rate
        self.theta[2] -= self.rate

    self.ee_pos = self.forward_kinematics(
        self.theta[0], self.theta[1], self.theta[2])[0]

    self.theta[0] = np.clip(self.theta[0], self.min_theta, self.max_theta)
    self.theta[1] = np.clip(self.theta[1], self.min_theta, self.max_theta)
    self.theta[2] = np.clip(self.theta[2], self.min_theta, self.max_theta)
```

```
        self.theta[0] = self.normalize_angle(self.theta[0])
        self.theta[1] = self.normalize_angle(self.theta[1])
        self.theta[2] = self.normalize_angle(self.theta[2])

        a = np.array((self.goal[0], self.goal[1], self.goal[2]))
        b = np.array((self.ee_pos[0], self.ee_pos[1], self.ee_pos[2]))

        self.distance = np.linalg.norm(a-b)
        #print("DISTANCIA DELTA ENV: ", distance)

        done = False
        reward = 0
        if self.distance >= self.current_distance:
            reward = -1

        epsilon = 50

        if (self.distance > -epsilon and self.distance < epsilon):
            reward = 1
            done = True

        self.current_distance = self.distance
        self.current_score += reward

        if self.current_score == -10 or self.current_score >= 10:
            done = True
        else:
            done = False

        observation = np.hstack(
            (self.goal_pos, self.ee_pos, self.theta, self.distance))
        info = {
            'distance': self.distance,
            'goal_position': self.goal_pos,
            'ee_position': self.ee_pos
        }
        #self.render(self.theta, self.goal_pos)
        # self.timerate.sleep()
        return observation, reward, done, info

    def generate_random_positions(self):
        angles = np.arange(-1.1, 1.1, 0.01).tolist()

        theta0 = random.choice(angles)
        theta1 = random.choice(angles)
        theta2 = random.choice(angles)

        self.goal = self.forward_kinematics(theta0, theta1, theta2)[0]

        return self.goal

    @staticmethod
    def normalize_angle(angle):
        return mt.atan2(mt.sin(angle), mt.cos(angle))

    def reset(self, ep):
        self.goal_pos = self.generate_random_positions()

        self.theta[0] = 0
        self.theta[1] = 0
        self.theta[2] = 0

        self.current_score = 0

        self.ee_pos = self.forward_kinematics(
            self.theta[0], self.theta[1], self.theta[2])[0]
        observation = np.hstack(
            (self.goal_pos, self.ee_pos, self.theta, self.distance))

        return observation
```

```
        def render(self, theta, goal_pos):
            data = self.forward_kinematics(theta[0], theta[1], theta[2])

            ee_pos = data[0]
            ponto_1 = data[1]
            ponto_2 = data[2]
            ponto_3 = data[3]
            fig = plt.figure(1, figsize=(5, 5))

            ax = fig.add_subplot(111, projection='3d')

            # plot the point (2,3,4) on the figure
            ax.scatter(ee_pos[0], ee_pos[1], ee_pos[2], c='black', marker='o')
            # plot the point (2,3,4) on the figure
            ax.scatter(ponto_1[0], ponto_1[1], ponto_1[2], c='blue', marker='$O$')
            # plot the point (2,3,4) on the figure
            ax.scatter(ponto_2[0], ponto_2[1], ponto_2[2], c='blue', marker='$O$')
            # plot the point (2,3,4) on the figure
            ax.scatter(ponto_3[0], ponto_3[1], ponto_3[2], c='blue', marker='$O$')
            ax.scatter(goal_pos[0], goal_pos[1],
                       goal_pos[2], c='red', marker='$X$')

            t = np.linspace(0, 2*np.pi, 1000)
            z_line = ee_pos[2]  # np.linspace(0, 15, 1000)
            x_line = ee_pos[0] + 100*np.cos(t)
            y_line = ee_pos[1] + 100*np.sin(t)

            z_line1 = 0  # np.linspace(0, 15, 1000)
            x_line1 = 180*np.cos(t)
            y_line1 = 180*np.sin(t)

            ax.plot3D(x_line, y_line, z_line, 'blue')
            ax.plot3D(x_line1, y_line1, z_line1, 'red')

            x = [0,    ponto_1[0], 180 *
                 np.sin(np.deg2rad(60)), ponto_2[0], -180*np.sin(np.deg2rad(60)), ponto_3[0]]
            y = [-180,    ponto_1[1], 180 *
                 np.cos(np.deg2rad(60)), ponto_2[1],  180*np.cos(np.deg2rad(60)), ponto_3[1]]
            z = [0,    ponto_1[2], 0, ponto_2[2],
                 0,                        ponto_3[2]]

            x1 = [0 + ee_pos[0],    ponto_1[0], 100*np.sin(np.deg2rad(
                60))  +  ee_pos[0],  ponto_2[0],  -100*np.sin(np.deg2rad(60))  +  ee_pos[0],
ponto_3[0]]
            y1 = [-100 + ee_pos[1],    ponto_1[1], 100*np.cos(np.deg2rad(
                60))  +  ee_pos[1],  ponto_2[1],    100*np.cos(np.deg2rad(60))  +  ee_pos[1],
ponto_3[1]]
            z1 = [ee_pos[2],    ponto_1[2],                                ee_pos[2],
                  ponto_2[2],                                 ee_pos[2], ponto_3[2]]

            plt.plot(x, y, z,  'ro')

            def connectpoints(x, y, z, p1, p2):
                x1, x2 = x[p1], x[p2]
                y1, y2 = y[p1], y[p2]
                z1, z2 = z[p1], z[p2]
                plt.plot([x1, x2], [y1, y2], [z1, z2], 'k-')

            connectpoints(x, y, z, 0, 1)
            connectpoints(x, y, z, 2, 3)
            connectpoints(x, y, z, 4, 5)

            connectpoints(x1, y1, z1, 0, 1)
            connectpoints(x1, y1, z1, 2, 3)
            connectpoints(x1, y1, z1, 4, 5)

            ax.view_init(elev=30, azim=60)

            ax.axes.set_xlim3d(left=-650, right=650)
            ax.axes.set_ylim3d(bottom=-650, top=650)
            ax.axes.set_zlim3d(bottom=-1200, top=100)
```

```
            self.render_counter += 1
            plt.savefig(
                f"{RENDERS_DIR}/step_{self.render_counter}")
            ax.cla()
            plt.close(fig)

    !pip install tensorboardX

    from matplotlib import pyplot as plt
    import numpy as np
    from torch.utils.tensorboard import SummaryWriter

    if __name__ == "__main__":

        env = DeltaEnv()
        n_states = env.observation_space.n
        n_actions = env.action_space.n
        agent = DQNAgent(alpha=0.0001, n_states=n_states, n_actions=n_actions)
        writer = SummaryWriter(f'./log/{LOGS_DIR}')
        load_models = True
        n_episodes = NUMBER_OF_EPISODES
        n_steps = 500

        # Load weights
        if load_models:
            agent.eps = agent.eps_min
            agent.load_models()

        total_reward_hist = []
        avg_reward_hist = []
        for episode in range(1, n_episodes + 1):
            state = env.reset(episode)
            total_reward = 0
            for t in range(n_steps):
                # Render after episode 1800
                if episode > EPISODE_TO_START_PRINTING:
                    env.render(env.theta, env.goal_pos)
                action = agent.epsilon_greedy(state)
                next_state, reward, done, info = env.step(action)
                # print(info)
                # agent.learn(state, action, reward, next_state, done)
                state = next_state
                total_reward += reward
                if done:
                    break
            # if not load_models:
            #     agent.decrement_epsilon()

            # Save model
            # if episode > 100 and episode % 20 == 0:
            #     agent.save_models()

            # env.render()

            total_reward_hist.append(total_reward)
            avg_reward = np.average(total_reward_hist[-100:])
            avg_reward_hist.append(avg_reward)
            print("Episode :", episode, "Epsilon : {:.4f}".format(agent.eps), "Total Reward :
{:.4f}".format(
                total_reward), "Avg Reward : {:.4f}".format(avg_reward))
            with open(f"{REWARD_DATA_DIR}_REWARD_DATA.txt", "a") as myfile1:
                myfile1.write("%.4f, %.4f, %.4f, %.4f\n" % (episode, agent.eps, total_reward,
avg_reward))
            # Tensorboard log
            writer.add_scalar('Total Reward', total_reward, episode)
            writer.add_scalar('Avg Reward', avg_reward, episode)
            # writer.add_scalar('Epsilon', agent.eps, episode)
        fig, ax = plt.subplots(1)
        t = np.arange(n_episodes)
        ax.plot(t, total_reward_hist, label="Recompensas Totais")
```

```
        ax.plot(t, avg_reward_hist, label="Recompensa média")
        ax.set_title("Recompensas vs Episódios")
        ax.set_xlabel("Episódio")
        ax.set_ylabel("Recompensa")
        ax.legend()
        plt.savefig(f"{RESULTS_FIG_NAME}.png")
        ax.cla()
        plt.close(fig)

    # from matplotlib import pyplot as plt
    # import numpy as np
    # from tensorboardX import SummaryWriter

    # env = DeltaEnv()
    # n_states = env.observation_space.n
    # n_actions = env.action_space.n
    # agent = DQNAgent(alpha=0.0001, n_states=n_states, n_actions=n_actions)
    # writer = SummaryWriter(f'./log/{LOGS_DIR}')
    # load_models = True
    # n_episodes = NUMBER_OF_EPISODES
    # n_steps = 500

    # # Load weights
    # if load_models:
    #     agent.eps = agent.eps_min
    #     agent.load_models()

    # total_reward_hist = []
    # avg_reward_hist = []
    # for episode in range(1, n_episodes + 1):
    #     state = env.reset(episode)
    #     total_reward = 0
    #     if episode > EPISODE_TO_START_PRINTING:
    #         env.render(env.theta, env.goal_pos)
    #         action = agent.epsilon_greedy(state)
    #         next_state, reward, done, info = env.step(action)
    #         # print(info)
    #         # agent.learn(state, action, reward, next_state, done)
    #         state = next_state
    #         total_reward += reward
    #         print(total_reward)
    #         if done:
    #             break
    #     if not load_models:
    #         agent.decrement_epsilon()

    #     # Save model
    #     # if episode > 100 and episode % 20 == 0:
    #     #     agent.save_models()

    #     # env.render()

    #     total_reward_hist.append(total_reward)
    #     avg_reward = np.average(total_reward_hist[-100:])
    #     avg_reward_hist.append(avg_reward)
    #      print("Episode :", episode, "Epsilon : {:.2f}".format(agent.eps), "Total Reward :
{:.2f}".format(
    #         total_reward), "Avg Reward : {:.2f}".format(avg_reward))
    #     with open(f"{REWARD_DATA_DIR}_REWARD_DATA.txt", "a") as myfile1:
    #          myfile1.write("%.4f, %.4f, %.4f, %.4f\n" % (episode, agent.eps, total_reward,
avg_reward))
    #     # Tensorboard log
    #     writer.add_scalar('Total Reward', total_reward, episode)
    #     writer.add_scalar('Avg Reward', avg_reward, episode)
    #     # writer.add_scalar('Epsilon', agent.eps, episode)
    # fig, ax = plt.subplots(1)
    # t = np.arange(n_episodes)
    # ax.plot(t, total_reward_hist, label="Recompensas Totais")
    # ax.plot(t, avg_reward_hist, label="Recompensa média")
    # ax.set_title("Recompensas vs Episódios")
    # ax.set_xlabel("Episódio")
```

```
    # ax.set_ylabel("Recompensa")
    # ax.legend()
    # plt.savefig(f"{RESULTS_FIG_NAME}.png")
    # ax.cla()
    # plt.close(fig)

    # from matplotlib import pyplot as plt
    # import numpy as np
    # from torch.utils.tensorboard import SummaryWriter

    # if __name__ == "__main__":

    #     env = DeltaEnv()
    #     n_states = env.observation_space.n
    #     n_actions = env.action_space.n
    #     agent = DQNAgent(alpha=0.0001, n_states=n_states, n_actions=n_actions)
    #     writer = SummaryWriter(f'./log/{LOGS_DIR}')
    #     load_models = True
    #     n_episodes = NUMBER_OF_EPISODES
    #     n_steps = 500

    #     # Load weights
    #     if load_models:
    #         agent.eps = agent.eps_min
    #         agent.load_models()

    #     total_reward_hist = []
    #     avg_reward_hist = []
    #     for episode in range(1, n_episodes + 1):
    #         state = np.array(env.reset())
    #         total_reward = 0
    #         for t in range(n_steps):
    #             # Render after episode 1800
    #             if episode > EPISODE_TO_START_PRINTING:
    #                 env.render(env.theta, env.goal_pos)
    #             action = agent.epsilon_greedy(state)
    #             next_state, reward, done, info = env.step(action)
    #             # print(info)
    #             # agent.learn(state, action, reward, next_state, done)
    #             state = next_state
    #             total_reward += reward
    #             if done:
    #                 break
    #         if not load_models:
    #             agent.decrement_epsilon()

    #         # Save model
    #         # if episode > 100 and episode % 20 == 0:
    #         #     agent.save_models()

    #         # env.render()

    #         total_reward_hist.append(total_reward)
    #         avg_reward = np.average(total_reward_hist[-100:])
    #         avg_reward_hist.append(avg_reward)
    #         print("Episode :", episode, "Epsilon : {:.4f}".format(agent.eps), "Total Reward
: {:.4f}".format(
    #             total_reward), "Avg Reward : {:.4f}".format(avg_reward))
    #         with open(f"{REWARD_DATA_DIR}_REWARD_DATA.txt", "a") as myfile1:
    #            myfile1.write("%.4f, %.4f, %.4f, %.4f\n" % (episode, agent.eps, total_reward,
avg_reward))
    #         # Tensorboard log
    #         writer.add_scalar('Total Reward', total_reward, episode)
    #         writer.add_scalar('Avg Reward', avg_reward, episode)
    #         # writer.add_scalar('Epsilon', agent.eps, episode)
    #     fig, ax = plt.subplots(1)
    #     t = np.arange(n_episodes)
    #     ax.plot(t, total_reward_hist, label="Recompensas Totais")
    #     ax.plot(t, avg_reward_hist, label="Recompensa média")
    #     ax.set_title("Recompensas vs Episódios")
    #     ax.set_xlabel("Episódio")
```

```
    #     ax.set_ylabel("Recompensa")
    #     ax.legend()
    #     plt.savefig(f"{RESULTS_FIG_NAME}.png")
    #     ax.cla()
    #     plt.close(fig)

    file_path      =      DRIVE_PATH      +      "/REWARDS/"+      "REWARD_DATA_DIR_2022-08-
20_LR_=_0.0001_EPS_DECAY_=_0.999_EPS_MIN_=_0.01_N_EPISODES_=_2302_REWARD_DATA.txt"
    print(DRIVE_PATH           +           "/REWARDS/"+           "REWARD_DATA_DIR_2022-07-
31_LR_=_0.0001_EPS_DECAY_=_0.999_EPS_MIN_=_0.01_N_EPISODES_=_2302_REWARD_DATA")

    with open(f"{REWARD_DATA_DIR}_REWARD_DATA.txt", "r") as myfile1:
        file1 = myfile1.readlines()
        for line in file1:
            line


    file1 = open(file_path, 'r')
    Lines = file1.readlines()

    count = 0
    # Strips the newline character
    for line in Lines:
        count += 1
        print("{}".format(line.strip()))

    import pandas as pd

    file1 = pd.read_csv(file_path)
    file1.describe()

    import pandas as pd

    file1 = pd.read_csv(file_path)
    file1.describe()
    file1.to_numpy
    count = 0

    numpyfile1 = file1.to_numpy()
    print(numpyfile1)
    for reward in numpyfile1:
        print(reward)
        if(reward[2] == 10):
            count+=1
    count
    # import pandas as pd

    # file1 = pd.read_csv(file_path)
    # file1.to_numpy
    # count = 0

    # numpyfile1 = file1.to_numpy()
    # print(numpyfile1)
    # for reward in numpyfile1:
    #     print(reward)
    #     if(reward[2] == 10):
    #         count+=1
    # count

    # import pandas as pd

    #     file_path     =     DRIVE_PATH     +     "/REWARDS/"+     "REWARD_DATA_DIR_2022-07-
31_LR_=_0.0001_EPS_DECAY_=_0.999_EPS_MIN_=_0.001_N_EPISODES_=_6904_REWARD_DATA.txt"
    #         print(DRIVE_PATH         +         "/REWARDS/"+         "REWARD_DATA_DIR_2022-07-
31_LR_=_0.0001_EPS_DECAY_=_0.999_EPS_MIN_=_0.001_N_EPISODES_=_6904_REWARD_DATA")

    # file1 = pd.read_csv(file_path)
    # file1.to_numpy
    # count = 0

    # numpyfile1 = file1.to_numpy()
```

```
# print(numpyfile1)
# for reward in numpyfile1:
#     print(reward)
#     if(reward[2] == 10):
#         count+=1
# count
import numpy as np
import math as mt

def forward_kinematics(t1, t2, t3):
        '''
            Esta função recebe como entrada angulos: theta_1, theta_2, theta_3 e
            retorna o ponto no espaço no qual o efetuador do robô deve se
            posicionar.
        '''
        X, Y, Z = 0, 0, 0

        L = 400  # mm
        l = 900
        rA = 180
        rE = 100

        # t1 = np.deg2rad(t1)
        # t2 = np.deg2rad(t2)
        # t3 = np.deg2rad(t3)

        phi = np.deg2rad(30)
        r = rA - rE
        x1 = 0
        y1 = - (r + L * mt.cos(t1))
        z1 = - L * mt.sin(t1)
        ponto_1 = np.array((x1, y1, z1))

        x2 = (r + L * mt.cos(t2)) * mt.cos(phi)
        y2 = (r + L * mt.cos(t2)) * mt.sin(phi)
        z2 = - L * mt.sin(t2)
        ponto_2 = np.array((x2, y2, z2))

        x3 = - (r + L * mt.cos(t3)) * mt.cos(phi)
        y3 = (r + L * mt.cos(t3)) * mt.sin(phi)
        z3 = - L * mt.sin(t3)
        ponto_3 = np.array((x3, y3, z3))
        p1 = y1**2 + z1**2
        p2 = x2**2 + y2**2 + z2**2
        p3 = x3**2 + y3**2 + z3**2
        a1 = (z2 - z1) * (y3 - y1) - (z3 - z1) * (y2 - y1)
        b1 = - ((p2 - p1) * (y3 - y1) - (p3 - p1) * (y2 - y1)) /
        a2 = - (z2 - z1) * x3 + (z3 - z1) * x2
        b2 = ((p2 - p1) * x3 - (p3 - p1) * x2) / 2
        dnm = (y2 - y1) * x3 - (y3 - y1) * x2
        a = a1**2 + a2**2 + dnm**2
        b = 2 * (a1 * b1 + a2 * (b2 - y1 * dnm) - z1 * dnm**2)
        c = (b2 - y1 * dnm) * (b2 - y1 * dnm) + \
            b1**2 + (dnm**2) * (z1**2 - l**2)
        d = b * b - 4.0 * a * c
        if (d < 0):
            Z = -1
            b + mt.sqrt(-d)
            b
            a
        else:
            Z = - 0.5 * (b + mt.sqrt(d)) / a
            X = (a1 * Z + b1) / dnm
            Y = (a2 * Z + b2) / dnm

        ee_pos = np.array((X, Y, Z))

        return ee_pos, ponto_1, ponto_2, ponto_3

np.round(forward_kinematics(-45, -45, -45)[0])
```

ANEXO D – CÓDIGO PARA TESTAR PONTOS FIXOS

```
    # -*- coding: utf-8 -*-
    """TCC_JULHO_DE_2022_-_TESTS_FIXED_GOALS

    Automatically generated by Colaboratory.

    Original file is located at
        https://colab.research.google.com/drive/1XpWkEK83gi-5pFYqCIvMhd8GvHu3mWQV
    """

    from google.colab import drive
    drive.mount('/drive/')

    import datetime
    import os

    # 0.1*Power[0.999,x]=0.01
    DAY = str(datetime.date.today())
    LEARNING_RATE = 0.0001
    EPS = 0.1
    EPS_DECAY = 0.999
    EPS_MIN = 0.01
    NUMBER_OF_EPISODES = 1000
    EPISODE_TO_START_PRINTING = NUMBER_OF_EPISODES - 10
    PONTO = 18
    DRIVE_PATH                                                                                  =
f"/drive/MyDrive/TCC_JULHO_DE_2022/1000_EPISODES_FIXED_GOALS_TESTS/TESTS_FIXED_GOALS_WITH_DI
STANCE_DATA/PONTO_{PONTO}_NOVO_TESTE_{DAY}/DISTANCES"
    RUN_PARAMETERS                                                                              =
f"_LR_=_{LEARNING_RATE}_EPS_DECAY_=_{EPS_DECAY}_EPS_MIN_=_{EPS_MIN}_N_EPISODES_=_{NUMBER_OF_
EPISODES}"
    WEIGHTS_DIR  =  "/drive/MyDrive/TCC_JULHO_DE_2022/TRAININGS/WEIGHTS/WEIGHTS_DIR_2022-08-
20_LR_=_0.0001_EPS_DECAY_=_0.999_EPS_MIN_=_0.01_N_EPISODES_=_2302"     #     DRIVE_PATH     +
"/WEIGHTS/WEIGHTS_DIR_" + DAY + RUN_PARAMETERS
    RENDERS_DIR =  DRIVE_PATH + "/RENDERS/RENDERS_DIR_"  + DAY + RUN_PARAMETERS
    LOGS_DIR = DRIVE_PATH + "/LOGS/LOGS_DIR_" + DAY + RUN_PARAMETERS
    REWARD_DATA_DIR = DRIVE_PATH + "/REWARDS/REWARD_DATA_DIR_" + DAY + RUN_PARAMETERS
    RESULTS_FIG_NAME = DRIVE_PATH + "/RESULTS/RESULTS_FIG_"  + DAY + RUN_PARAMETERS

    WEIGHTS_DIR

    import os

    dir_list   =   [DRIVE_PATH,   WEIGHTS_DIR,   RENDERS_DIR,   LOGS_DIR,   REWARD_DATA_DIR,
RESULTS_FIG_NAME]

    # Check whether the specified path exists or not
    for dir in dir_list:
        isExist = os.path.exists(dir)

        if not isExist:

            # Create a new directory because it does not exist
            os.makedirs(dir)
            print("The new directory is created!")

    import os
    import torch as T
    import torch.nn as nn
    import torch.optim as optim
    import torch.nn.functional as F


    class QNetwork(nn.Module):
        def       __init__(self,       lr=LEARNING_RATE,       n_states=4,       n_actions=6,
checkpoint_dir=f"./{WEIGHTS_DIR}", filename=f"{WEIGHTS_DIR}"):
            super(QNetwork, self).__init__()
            if not os.path.isdir(checkpoint_dir):
                os.makedirs(checkpoint_dir)
```

```
        self.checkpoint_file = os.path.join(checkpoint_dir, filename)
        # Detalhe da rede neural
        self.fc1 = nn.Linear(n_states, 300)
        self.fc2 = nn.Linear(300, 400)
        self.fc3 = nn.Linear(400, 600)
        self.fc4 = nn.Linear(600, 600)
        self.fc5 = nn.Linear(600, 400)
        self.fc6 = nn.Linear(400, 300)
        self.fc7 = nn.Linear(300, n_actions)
        # Optimizer e loss
        self.optimizer = optim.Adam(self.parameters(), lr=lr)
        self.loss = nn.MSELoss()
        self.device = T.device('cuda:0' if T.cuda.is_available() else 'cpu')
        self.to(self.device)

    def forward(self, state):
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        x = F.relu(self.fc4(x))
        x = F.relu(self.fc5(x))
        x = F.relu(self.fc6(x))
        return self.fc7(x)

    def save_checkpoint(self):
        print('... Save checkpoint ...')
        T.save(self.state_dict(), self.checkpoint_file)

    def load_checkpoint(self):
        print('... Load checkpoint ...')
        self.load_state_dict(T.load(self.checkpoint_file))

import numpy as np


class ReplayMemory(object):
    def __init__(self, max_size=10000, n_states=4):
        self.max_size = max_size
        self.memory_counter = 0
        self.states_memory = np.zeros((max_size, n_states), dtype=np.float32)
        self.next_states_memory = np.zeros(
            (max_size, n_states), dtype=np.float32)
        self.actions_memory = np.zeros(max_size, dtype=np.int64)
        self.rewards_memory = np.zeros(max_size, dtype=np.float32)
        self.dones_memory = np.zeros(max_size, dtype=bool)

    def store_transition(self, state, action, reward, next_state, done):
        index = self.memory_counter % self.max_size
        self.states_memory[index] = state
        self.actions_memory[index] = action
        self.rewards_memory[index] = reward
        self.next_states_memory[index] = next_state
        self.dones_memory[index] = done
        self.memory_counter += 1

    def sample_memory(self, batch_size):
        max_mem = min(self.memory_counter, self.max_size)
        batch = np.random.choice(max_mem, batch_size, replace=False)
        states = self.states_memory[batch]
        actions = self.actions_memory[batch]
        rewards = self.rewards_memory[batch]
        next_states = self.next_states_memory[batch]
        dones = self.dones_memory[batch]
        return states, actions, rewards, next_states, dones

np.zeros(10000, dtype=np.int64)

import numpy as np
import torch as T

class DQNAgent(object):
```

```
        def __init__(
            self,
            alpha=0.0005,
            gamma=0.99,
            eps=EPS,
            eps_decay=EPS_DECAY,
            eps_min=EPS_MIN,
            tau=0.001,
            max_size=100000,
            batch_size=64,
            update_rate=4,
            n_states=0,
            n_actions=0,
            checkpoint_dir=f"{WEIGHTS_DIR}",
        ):

            self.gamma = gamma
            self.eps = eps
            self.eps_decay = eps_decay
            self.eps_min = eps_min
            self.tau = tau
            self.batch_size = batch_size
            self.update_rate = update_rate
            self.action_space = [i for i in range(n_actions)]

            # Replay memory
            self.memory = ReplayMemory(max_size=max_size, n_states=n_states)

            # Q-Network
            self.qnetwork_local = QNetwork(lr=alpha, n_states=n_states, n_actions=n_actions,
                                           checkpoint_dir=checkpoint_dir,
filename="qnetwork_local_TESTE.pth")
            self.qnetwork_target = QNetwork(lr=alpha, n_states=n_states, n_actions=n_actions,
                                            checkpoint_dir=checkpoint_dir,
filename="qnetwork_target_TESTE.pth")

            self.counter = 0

        def decrement_epsilon(self):
            self.eps *= self.eps_decay
            if self.eps < self.eps_min:
                self.eps = self.eps_min

        def epsilon_greedy(self, state):
            if np.random.random() > self.eps:
                state = T.tensor([state], dtype=T.float).to(
                    self.qnetwork_local.device)
                actions = self.qnetwork_local.forward(state)
                action = T.argmax(actions).item()
            else:
                action = np.random.choice(self.action_space)
            return action

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

        def sample_memory(self):
            states, actions, rewards, next_states, dones = \
                self.memory.sample_memory(self.batch_size)
            t_states = T.tensor(states).to(self.qnetwork_local.device)
            t_actions = T.tensor(actions).to(self.qnetwork_local.device)
            t_rewards = T.tensor(rewards).to(self.qnetwork_local.device)
            t_next_states = T.tensor(next_states).to(self.qnetwork_local.device)
            t_dones = T.tensor(dones).to(self.qnetwork_local.device)
            return t_states, t_actions, t_rewards, t_next_states, t_dones

        def save_models(self):
            self.qnetwork_local.save_checkpoint()
            self.qnetwork_target.save_checkpoint()

        def load_models(self):
```

```
            self.qnetwork_local.load_checkpoint()
            self.qnetwork_target.load_checkpoint()

        def learn(self, state, action, reward, next_state, done):
            # Save experience to memory
            self.store_transition(state, action, reward, next_state, done)

            # If not enough memory then skip learning
            if self.memory.memory_counter < self.batch_size:
                return

            # Update target network parameter every update rate
            if self.counter % self.update_rate == 0:
                self.soft_update(self.tau)

            # Take random sampling from memory
            states, actions, rewards, next_states, dones = self.sample_memory()

            # Update action value
            indices = np.arange(self.batch_size)
            q_pred = self.qnetwork_local.forward(states)[indices, actions]
            q_next = self.qnetwork_target.forward(next_states).max(dim=1)[0]
            q_next[dones] = 0.0
            q_target = rewards + self.gamma*q_next
            self.qnetwork_local.optimizer.zero_grad()
            loss = \
                self.qnetwork_local.loss(q_target, q_pred) \
                .to(self.qnetwork_local.device)
            loss.backward()
            self.qnetwork_local.optimizer.step()
            self.counter += 1

        def soft_update(self, tau):
            for   target_param,   local_param   in   zip(self.qnetwork_target.parameters(),
self.qnetwork_local.parameters()):
                target_param.data.copy_(
                    tau*local_param.data + (1.0-tau)*target_param.data)

        def regular_update(self):
            for   target_param,   local_param   in   zip(self.qnetwork_target.parameters(),
self.qnetwork_local.parameters()):
                target_param.data.copy_(local_param.data + target_param.data)

    from mpl_toolkits.mplot3d import Axes3D
    import os
    import random
    import numpy as np
    import time
    from gym import spaces
    import math as mt
    import matplotlib.pyplot as plt
    import matplotlib
    matplotlib.use('Agg')


    class DeltaEnv():
        def __init__(self):

            self.min_theta = -1.1
            self.max_theta = 1.1
            self.theta = np.array([0.0, 0.0, 0.0])

            self.distance = 1000
            self.current_distance = 1000

            self.set_increment_rate(0.1)
            self.render_counter = 0

            self.action = {
                0: "HOLD",
                1: "INC_J1",
```

```
            2: "DEC_J1",
            3: "INC_J2",
            4: "DEC_J2",
            5: "INC_J3",
            6: "DEC_J3",
            7: "INC_J1_J2",
            8: "DEC_J1_J2",
            9: "INC_J2_J3",
            10: "DEC_J2_J3",
            11: "INC_J1_J3",
            12: "DEC_J1_J3",
            13: "INC_J1_J2_J3",
            14: "DEC_J1_J2_J3"}

        self.ee_pos = self.forward_kinematics(
            self.theta[0], self.theta[1], self.theta[2])[0]
        self.goal_pos = self.generate_random_positions()[0]
        self.states = np.hstack(
            (self.goal_pos, self.ee_pos, self.theta, self.distance, 0, 0))

        self.action_space = spaces.Discrete(len(self.action))
        self.observation_space = spaces.Discrete(len(self.states))

    def forward_kinematics(self, t1, t2, t3):
        '''
            Esta função recebe como entrada angulos: theta_1, theta_2, theta_3 e
            retorna o ponto no espaço no qual o efetuador do robô deve se
            posicionar.
        '''
        X, Y, Z = 0, 0, 0

        L = 400  # mm
        l = 900
        rA = 180
        rE = 100

        # t1 = np.deg2rad(t1)
        # t2 = np.deg2rad(t2)
        # t3 = np.deg2rad(t3)

        phi = np.deg2rad(30)
        r = rA - rE

        x1 = 0
        y1 = - (r + L * mt.cos(t1))
        z1 = - L * mt.sin(t1)
        ponto_1 = np.array((x1, y1, z1))

        x2 = (r + L * mt.cos(t2)) * mt.cos(phi)
        y2 = (r + L * mt.cos(t2)) * mt.sin(phi)
        z2 = - L * mt.sin(t2)
        ponto_2 = np.array((x2, y2, z2))

        x3 = - (r + L * mt.cos(t3)) * mt.cos(phi)
        y3 = (r + L * mt.cos(t3)) * mt.sin(phi)
        z3 = - L * mt.sin(t3)
        ponto_3 = np.array((x3, y3, z3))

        p1 = y1**2 + z1**2
        p2 = x2**2 + y2**2 + z2**2
        p3 = x3**2 + y3**2 + z3**2

        a1 = (z2 - z1) * (y3 - y1) - (z3 - z1) * (y2 - y1)
        b1 = - ((p2 - p1) * (y3 - y1) - (p3 - p1) * (y2 - y1)) / 2

        a2 = - (z2 - z1) * x3 + (z3 - z1) * x2
        b2 = ((p2 - p1) * x3 - (p3 - p1) * x2) / 2

        dnm = (y2 - y1) * x3 - (y3 - y1) * x2

        a = a1**2 + a2**2 + dnm**2
```

```
    b = 2 * (a1 * b1 + a2 * (b2 - y1 * dnm) - z1 * dnm**2)
    c = (b2 - y1 * dnm) * (b2 - y1 * dnm) + \
        b1**2 + (dnm**2) * (z1**2 - l**2)

    d = b * b - 4.0 * a * c

    if (d < 0):
        Z = -1
        b + mt.sqrt(-d)
        b
        a
    else:
        Z = - 0.5 * (b + mt.sqrt(d)) / a
        X = (a1 * Z + b1) / dnm
        Y = (a2 * Z + b2) / dnm

    ee_pos = np.array((X, Y, Z))

    return ee_pos, ponto_1, ponto_2, ponto_3

def set_increment_rate(self, rate):
    self.rate = rate

def step(self, action):
    if self.action[action] == "HOLD":
        self.theta[0] += 0  # self.rate
        self.theta[1] += 0  # self.rate
        self.theta[2] += 0  # self.rate
    elif self.action[action] == "INC_J1":
        self.theta[0] += self.rate
    elif self.action[action] == "DEC_J1":
        self.theta[0] -= self.rate
    elif self.action[action] == "INC_J2":
        self.theta[1] += self.rate
    elif self.action[action] == "DEC_J2":
        self.theta[1] -= self.rate
    elif self.action[action] == "INC_J3":
        self.theta[2] += self.rate
    elif self.action[action] == "DEC_J3":
        self.theta[2] -= self.rate
    elif self.action[action] == "INC_J1_J2":
        self.theta[0] += self.rate
        self.theta[1] += self.rate
    elif self.action[action] == "DEC_J1_J2":
        self.theta[0] -= self.rate
        self.theta[1] -= self.rate
    elif self.action[action] == "INC_J2_J3":
        self.theta[1] += self.rate
        self.theta[2] += self.rate
    elif self.action[action] == "DEC_J2_J3":
        self.theta[1] -= self.rate
        self.theta[2] -= self.rate
    elif self.action[action] == "INC_J1_J3":
        self.theta[0] += self.rate
        self.theta[2] += self.rate
    elif self.action[action] == "DEC_J1_J3":
        self.theta[0] -= self.rate
        self.theta[2] -= self.rate
    elif self.action[action] == "INC_J1_J2_J3":
        self.theta[0] += self.rate
        self.theta[1] += self.rate
        self.theta[2] += self.rate
    elif self.action[action] == "DEC_J1_J2_J3":
        self.theta[0] -= self.rate
        self.theta[1] -= self.rate
        self.theta[2] -= self.rate

    self.ee_pos = self.forward_kinematics(
        self.theta[0], self.theta[1], self.theta[2])[0]

    self.theta[0] = np.clip(self.theta[0], self.min_theta, self.max_theta)
```

```
        self.theta[1] = np.clip(self.theta[1], self.min_theta, self.max_theta)
        self.theta[2] = np.clip(self.theta[2], self.min_theta, self.max_theta)

        self.theta[0] = self.normalize_angle(self.theta[0])
        self.theta[1] = self.normalize_angle(self.theta[1])
        self.theta[2] = self.normalize_angle(self.theta[2])

        a = np.array((self.goal[0], self.goal[1], self.goal[2]))
        b = np.array((self.ee_pos[0], self.ee_pos[1], self.ee_pos[2]))

        self.distance = np.linalg.norm(a-b)
        #print("DISTANCIA DELTA ENV: ", distance)

        done = False
        reward = 0
        if self.distance >= self.current_distance:
            reward = -1

        epsilon = 50

        if (self.distance > -epsilon and self.distance < epsilon):
            reward = 1
            done = True

        self.current_distance = self.distance
        self.current_score += reward

        if self.current_score == -10 or self.current_score >= 10:
            done = True
        else:
            done = False

        observation = np.hstack(
            (self.goal_pos, self.ee_pos, self.theta, self.distance))
        info = [
            self.distance,
            self.goal_pos,
            self.ee_pos
        ]
        #self.render(self.theta, self.goal_pos)
        # self.timerate.sleep()
        return observation, reward, done, info

    def generate_random_positions(self):
        angles = np.arange(-1.1, 1.1, 0.01).tolist()

        theta0 = random.choice(angles)
        theta1 = random.choice(angles)
        theta2 = random.choice(angles)
        degree_angles = [
        [-45, -45, -45], # 1
        [0, -45, -45], # 2
        [0, 0, -45], # 3
        [45, -45, 0], # 4
        [30, 30, 0], # 5
        [-45, 25, -25],
        [-10, 30, -30], # 7
        [45, 45, 45],
        [-30, -30, 0],
        [-45, 25, -25],
        [30, 0, 45],
        [15, -15, 25],
        [10,-25, -15],
        [-35, -15, 5],
        [10, -30, 10],
        [-25, 10, -5],
        [-30, 5, 30],
        [-30, 45, 30],
        [-15, 10, 15],
        [-5, -10, 5],
        [-10, 30, 40],
```

```
            [-30, -30, -30],]

            radian_angles = []
            rangle = []

            for angle in degree_angles:
                rangle.append([np.deg2rad(angle[0]),                    np.deg2rad(angle[1]),
np.deg2rad(angle[2])])
            print(rangle)
            print(radian_angles)

            theta0 = rangle[PONTO - 1][0]
            theta1 = rangle[PONTO - 1][1]
            theta2 = rangle[PONTO - 1][2]

            self.goal = self.forward_kinematics(theta0, theta1, theta2)[0]

            return self.goal

        @staticmethod
        def normalize_angle(angle):
            return mt.atan2(mt.sin(angle), mt.cos(angle))

        def reset(self, ep):
            self.goal_pos = self.generate_random_positions()

            self.theta[0] = 0
            self.theta[1] = 0
            self.theta[2] = 0

            self.current_score = 0

            self.ee_pos = self.forward_kinematics(
                self.theta[0], self.theta[1], self.theta[2])[0]
            observation = np.hstack(
                (self.goal_pos, self.ee_pos, self.theta, self.distance))

            return observation

        def render(self, theta, goal_pos):
            data = self.forward_kinematics(theta[0], theta[1], theta[2])

            ee_pos = data[0]
            ponto_1 = data[1]
            ponto_2 = data[2]
            ponto_3 = data[3]
            fig = plt.figure(1, figsize=(5, 5))

            ax = fig.add_subplot(111, projection='3d')

            # plot the point (2,3,4) on the figure
            ax.scatter(ee_pos[0], ee_pos[1], ee_pos[2], c='black', marker='o')
            # plot the point (2,3,4) on the figure
            ax.scatter(ponto_1[0], ponto_1[1], ponto_1[2], c='blue', marker='$O$')
            # plot the point (2,3,4) on the figure
            ax.scatter(ponto_2[0], ponto_2[1], ponto_2[2], c='blue', marker='$O$')
            # plot the point (2,3,4) on the figure
            ax.scatter(ponto_3[0], ponto_3[1], ponto_3[2], c='blue', marker='$O$')
            ax.scatter(goal_pos[0], goal_pos[1],
                       goal_pos[2], c='red', marker='$X$')

            t = np.linspace(0, 2*np.pi, 1000)
            z_line = ee_pos[2]  # np.linspace(0, 15, 1000)
            x_line = ee_pos[0] + 100*np.cos(t)
            y_line = ee_pos[1] + 100*np.sin(t)

            z_line1 = 0  # np.linspace(0, 15, 1000)
            x_line1 = 180*np.cos(t)
            y_line1 = 180*np.sin(t)

            ax.plot3D(x_line, y_line, z_line, 'blue')
```

```
            ax.plot3D(x_line1, y_line1, z_line1, 'red')

            x = [0,    ponto_1[0], 180 *
                 np.sin(np.deg2rad(60)), ponto_2[0], -180*np.sin(np.deg2rad(60)), ponto_3[0]]
            y = [-180,    ponto_1[1], 180 *
                 np.cos(np.deg2rad(60)), ponto_2[1],  180*np.cos(np.deg2rad(60)), ponto_3[1]]
            z = [0,    ponto_1[2], 0, ponto_2[2],
                 0,                        ponto_3[2]]

            x1 = [0 + ee_pos[0],    ponto_1[0], 100*np.sin(np.deg2rad(
                60))  +  ee_pos[0],  ponto_2[0],  -100*np.sin(np.deg2rad(60))  +  ee_pos[0],
ponto_3[0]]
            y1 = [-100 + ee_pos[1],    ponto_1[1], 100*np.cos(np.deg2rad(
                60))  +  ee_pos[1],  ponto_2[1],    100*np.cos(np.deg2rad(60))  +  ee_pos[1],
ponto_3[1]]
            z1 = [ee_pos[2],    ponto_1[2],                              ee_pos[2],
                  ponto_2[2],                                ee_pos[2], ponto_3[2]]

            plt.plot(x, y, z,  'ro')

            def connectpoints(x, y, z, p1, p2):
                x1, x2 = x[p1], x[p2]
                y1, y2 = y[p1], y[p2]
                z1, z2 = z[p1], z[p2]
                plt.plot([x1, x2], [y1, y2], [z1, z2], 'k-')

            connectpoints(x, y, z, 0, 1)
            connectpoints(x, y, z, 2, 3)
            connectpoints(x, y, z, 4, 5)

            connectpoints(x1, y1, z1, 0, 1)
            connectpoints(x1, y1, z1, 2, 3)
            connectpoints(x1, y1, z1, 4, 5)

            ax.view_init(elev=30, azim=60)

            ax.axes.set_xlim3d(left=-650, right=650)
            ax.axes.set_ylim3d(bottom=-650, top=650)
            ax.axes.set_zlim3d(bottom=-1200, top=100)

            self.render_counter += 1
            plt.savefig(
                f"{RENDERS_DIR}/step_{self.render_counter}")
            ax.cla()
            plt.close(fig)

    !pip install tensorboardX

    from matplotlib import pyplot as plt
    import numpy as np
    from torch.utils.tensorboard import SummaryWriter

    if __name__ == "__main__":

        env = DeltaEnv()
        n_states = env.observation_space.n
        n_actions = env.action_space.n
        agent = DQNAgent(alpha=0.0001, n_states=n_states, n_actions=n_actions)
        writer = SummaryWriter(f'./log/{LOGS_DIR}')
        load_models = True
        n_episodes = NUMBER_OF_EPISODES
        n_steps = 500

        # Load weights
        if load_models:
            agent.eps = agent.eps_min
            agent.load_models()

        total_reward_hist = []
        avg_reward_hist = []
        for episode in range(1, n_episodes + 1):
```

```
            state = env.reset(episode)
            total_reward = 0
            for t in range(n_steps):
                # Render after episode 1800
                if episode > EPISODE_TO_START_PRINTING:
                    env.render(env.theta, env.goal_pos)
                action = agent.epsilon_greedy(state)
                next_state, reward, done, info = env.step(action)
                with open(f"{REWARD_DATA_DIR}_DISTANCE_DATA.txt", "a") as myfile2:
                    myfile2.write("%.4f, %.4f, %.4f\n" % (episode, t,  info[0]))
                with open(f"{REWARD_DATA_DIR}_DISTANCE_GOAL_ENDEFFECTOR_POSITIONS.txt", "a")
as myfile3:
                    myfile3.write("%.4f,  %.4f,  %.4f,  %.4f,  %.4f,  %.4f,  %.4f,  %.4f,  %.4f,
%.4f\n"  %  (episode,  t,  reward,  info[0],  info[1][0],  info[1][1],info[1][2],  info[2][0],
info[2][1], info[2][2]))
                # agent.learn(state, action, reward, next_state, done)
                state = next_state
                total_reward += reward
                if done:
                    break
            # if not load_models:
            #     agent.decrement_epsilon()

            # Save model
            # if episode > 100 and episode % 20 == 0:
            #     agent.save_models()

            # env.render()

            total_reward_hist.append(total_reward)
            avg_reward = np.average(total_reward_hist[-100:])
            avg_reward_hist.append(avg_reward)
            print("Episode :", episode, "Epsilon : {:.4f}".format(agent.eps), "Total Reward :
{:.4f}".format(
                total_reward), "Avg Reward : {:.4f}".format(avg_reward))
            with open(f"{REWARD_DATA_DIR}_REWARD_DATA.txt", "a") as myfile1:
                myfile1.write("%.4f, %.4f, %.4f, %.4f\n" % (episode, agent.eps, total_reward,
avg_reward))
            # Tensorboard log
            writer.add_scalar('Total Reward', total_reward, episode)
            writer.add_scalar('Avg Reward', avg_reward, episode)
            # writer.add_scalar('Epsilon', agent.eps, episode)
        fig, ax = plt.subplots(1)
        t = np.arange(n_episodes)
        ax.plot(t, total_reward_hist, label="Recompensas Totais")
        ax.plot(t, avg_reward_hist, label="Recompensa média")
        ax.set_title("Recompensas vs Episódios")
        ax.set_xlabel("Episódio")
        ax.set_ylabel("Recompensa")
        ax.legend()
        plt.savefig(f"{RESULTS_FIG_NAME}.png")
        ax.cla()
        plt.close(fig)

    # from matplotlib import pyplot as plt
    # import numpy as np
    # from tensorboardX import SummaryWriter

    # env = DeltaEnv()
    # n_states = env.observation_space.n
    # n_actions = env.action_space.n
    # agent = DQNAgent(alpha=0.0001, n_states=n_states, n_actions=n_actions)
    # writer = SummaryWriter(f'./log/{LOGS_DIR}')
    # load_models = True
    # n_episodes = NUMBER_OF_EPISODES
    # n_steps = 500

    # # Load weights
    # if load_models:
    #     agent.eps = agent.eps_min
    #     agent.load_models()
```

```
    # total_reward_hist = []
    # avg_reward_hist = []
    # for episode in range(1, n_episodes + 1):
    #     state = env.reset(episode)
    #     total_reward = 0
    #     if episode > EPISODE_TO_START_PRINTING:
    #         env.render(env.theta, env.goal_pos)
    #         action = agent.epsilon_greedy(state)
    #         next_state, reward, done, info = env.step(action)
    #         # print(info)
    #         # agent.learn(state, action, reward, next_state, done)
    #         state = next_state
    #         total_reward += reward
    #         print(total_reward)
    #         if done:
    #             break
    #     if not load_models:
    #         agent.decrement_epsilon()

    #     # Save model
    #     # if episode > 100 and episode % 20 == 0:
    #     #     agent.save_models()

    #     # env.render()

    #     total_reward_hist.append(total_reward)
    #     avg_reward = np.average(total_reward_hist[-100:])
    #     avg_reward_hist.append(avg_reward)
    #      print("Episode :", episode, "Epsilon : {:.2f}".format(agent.eps), "Total Reward :
{:.2f}".format(
    #         total_reward), "Avg Reward : {:.2f}".format(avg_reward))
    #     with open(f"{REWARD_DATA_DIR}_REWARD_DATA.txt", "a") as myfile1:
    #          myfile1.write("%.4f, %.4f, %.4f, %.4f\n" % (episode, agent.eps, total_reward,
avg_reward))
    #     # Tensorboard log
    #     writer.add_scalar('Total Reward', total_reward, episode)
    #     writer.add_scalar('Avg Reward', avg_reward, episode)
    #     # writer.add_scalar('Epsilon', agent.eps, episode)
    # fig, ax = plt.subplots(1)
    # t = np.arange(n_episodes)
    # ax.plot(t, total_reward_hist, label="Recompensas Totais")
    # ax.plot(t, avg_reward_hist, label="Recompensa média")
    # ax.set_title("Recompensas vs Episódios")
    # ax.set_xlabel("Episódio")
    # ax.set_ylabel("Recompensa")
    # ax.legend()
    # plt.savefig(f"{RESULTS_FIG_NAME}.png")
    # ax.cla()
    # plt.close(fig)

    # from matplotlib import pyplot as plt
    # import numpy as np
    # from torch.utils.tensorboard import SummaryWriter

    # if __name__ == "__main__":

    #     env = DeltaEnv()
    #     n_states = env.observation_space.n
    #     n_actions = env.action_space.n
    #     agent = DQNAgent(alpha=0.0001, n_states=n_states, n_actions=n_actions)
    #     writer = SummaryWriter(f'./log/{LOGS_DIR}')
    #     load_models = True
    #     n_episodes = NUMBER_OF_EPISODES
    #     n_steps = 500

    #     # Load weights
    #     if load_models:
    #         agent.eps = agent.eps_min
    #         agent.load_models()
```

```
    #     total_reward_hist = []
    #     avg_reward_hist = []
    #     for episode in range(1, n_episodes + 1):
    #         state = np.array(env.reset())
    #         total_reward = 0
    #         for t in range(n_steps):
    #             # Render after episode 1800
    #             if episode > EPISODE_TO_START_PRINTING:
    #                 env.render(env.theta, env.goal_pos)
    #             action = agent.epsilon_greedy(state)
    #             next_state, reward, done, info = env.step(action)
    #             # print(info)
    #             # agent.learn(state, action, reward, next_state, done)
    #             state = next_state
    #             total_reward += reward
    #             if done:
    #                 break
    #         if not load_models:
    #             agent.decrement_epsilon()

    #         # Save model
    #         # if episode > 100 and episode % 20 == 0:
    #         #     agent.save_models()

    #         # env.render()

    #         total_reward_hist.append(total_reward)
    #         avg_reward = np.average(total_reward_hist[-100:])
    #         avg_reward_hist.append(avg_reward)
    #         print("Episode :", episode, "Epsilon : {:.4f}".format(agent.eps), "Total Reward
: {:.4f}".format(
    #             total_reward), "Avg Reward : {:.4f}".format(avg_reward))
    #         with open(f"{REWARD_DATA_DIR}_REWARD_DATA.txt", "a") as myfile1:
    #            myfile1.write("%.4f, %.4f, %.4f, %.4f\n" % (episode, agent.eps, total_reward,
avg_reward))
    #         # Tensorboard log
    #         writer.add_scalar('Total Reward', total_reward, episode)
    #         writer.add_scalar('Avg Reward', avg_reward, episode)
    #         # writer.add_scalar('Epsilon', agent.eps, episode)
    #     fig, ax = plt.subplots(1)
    #     t = np.arange(n_episodes)
    #     ax.plot(t, total_reward_hist, label="Recompensas Totais")
    #     ax.plot(t, avg_reward_hist, label="Recompensa média")
    #     ax.set_title("Recompensas vs Episódios")
    #     ax.set_xlabel("Episódio")
    #     ax.set_ylabel("Recompensa")
    #     ax.legend()
    #     plt.savefig(f"{RESULTS_FIG_NAME}.png")
    #     ax.cla()
    #     plt.close(fig)

    file_path      =      DRIVE_PATH      +      "/REWARDS/"+      "REWARD_DATA_DIR_2022-08-
20_LR_=_0.0001_EPS_DECAY_=_0.999_EPS_MIN_=_0.01_N_EPISODES_=_2302_REWARD_DATA.txt"
    print(DRIVE_PATH           +           "/REWARDS/"+           "REWARD_DATA_DIR_2022-07-
31_LR_=_0.0001_EPS_DECAY_=_0.999_EPS_MIN_=_0.01_N_EPISODES_=_2302_REWARD_DATA")

    # with open(f"{REWARD_DATA_DIR}_REWARD_DATA.txt", "r") as myfile1:
    #     file1 = myfile1.readlines()
    #     for line in file1:
    #         line


    # file1 = open(file_path, 'r')
    # Lines = file1.readlines()

    # count = 0
    # # Strips the newline character
    # for line in Lines:
    #     count += 1
    #     print("{}".format(line.strip()))
```

```
    # import pandas as pd

    # file1 = pd.read_csv(file_path)
    # file1.describe()

    # import pandas as pd

    # file1 = pd.read_csv(file_path)
    # file1.describe()
    # file1.to_numpy
    # count = 0

    # numpyfile1 = file1.to_numpy()
    # print(numpyfile1)
    # for reward in numpyfile1:
    #     print(reward)
    #     if(reward[2] == 10):
    #         count+=1
    # count

    1958/2302

    # import pandas as pd

    # file1 = pd.read_csv(file_path)
    # file1.to_numpy
    # count = 0

    # numpyfile1 = file1.to_numpy()
    # print(numpyfile1)
    # for reward in numpyfile1:
    #     print(reward)
    #     if(reward[2] == 10):
    #         count+=1
    # count

    # import pandas as pd

    #     file_path     =     DRIVE_PATH     +     "/REWARDS/"+     "REWARD_DATA_DIR_2022-07-
31_LR_=_0.0001_EPS_DECAY_=_0.999_EPS_MIN_=_0.001_N_EPISODES_=_6904_REWARD_DATA.txt"
    #         print(DRIVE_PATH        +         "/REWARDS/"+         "REWARD_DATA_DIR_2022-07-
31_LR_=_0.0001_EPS_DECAY_=_0.999_EPS_MIN_=_0.001_N_EPISODES_=_6904_REWARD_DATA")


    # file1 = pd.read_csv(file_path)
    # file1.to_numpy
    # count = 0

    # numpyfile1 = file1.to_numpy()
    # print(numpyfile1)
    # for reward in numpyfile1:
    #     print(reward)
    #     if(reward[2] == 10):
    #         count+=1
    # count

    import numpy as np
    import math as mt

    def forward_kinematics(t1, t2, t3):
        '''
            Esta função recebe como entrada angulos: theta_1, theta_2, theta_3 e
            retorna o ponto no espaço no qual o efetuador do robô deve se
            posicionar.
        '''
        X, Y, Z = 0, 0, 0

        L = 400  # mm
        l = 900
        rA = 180
        rE = 100
```

```
    phi = np.deg2rad(30)
    r = rA - rE

    x1 = 0
    y1 = - (r + L * mt.cos(t1))
    z1 = - L * mt.sin(t1)
    ponto_1 = np.array((x1, y1, z1))

    x2 = (r + L * mt.cos(t2)) * mt.cos(phi)
    y2 = (r + L * mt.cos(t2)) * mt.sin(phi)
    z2 = - L * mt.sin(t2)
    ponto_2 = np.array((x2, y2, z2))

    x3 = - (r + L * mt.cos(t3)) * mt.cos(phi)
    y3 = (r + L * mt.cos(t3)) * mt.sin(phi)
    z3 = - L * mt.sin(t3)
    ponto_3 = np.array((x3, y3, z3))

    p1 = y1**2 + z1**2
    p2 = x2**2 + y2**2 + z2**2
    p3 = x3**2 + y3**2 + z3**2

    a1 = (z2 - z1) * (y3 - y1) - (z3 - z1) * (y2 - y1)
    b1 = - ((p2 - p1) * (y3 - y1) - (p3 - p1) * (y2 - y1)) / 2

    a2 = - (z2 - z1) * x3 + (z3 - z1) * x2
    b2 = ((p2 - p1) * x3 - (p3 - p1) * x2) / 2

    dnm = (y2 - y1) * x3 - (y3 - y1) * x2

    a = a1**2 + a2**2 + dnm**2
    b = 2 * (a1 * b1 + a2 * (b2 - y1 * dnm) - z1 * dnm**2)
    c = (b2 - y1 * dnm) * (b2 - y1 * dnm) + \
        b1**2 + (dnm**2) * (z1**2 - l**2)

    d = b * b - 4.0 * a * c

    if (d < 0):
        Z = -1
        b + mt.sqrt(-d)
        b
        a
    else:
        Z = - 0.5 * (b + mt.sqrt(d)) / a
        X = (a1 * Z + b1) / dnm
        Y = (a2 * Z + b2) / dnm

    ee_pos = np.array((X, Y, Z))

    return ee_pos, ponto_1, ponto_2, ponto_3

degree_angles = [
[-45, -45, -45], # 1
[0, -45, -45], # 2
[0, 0, -45], # 3
[45, -45, 0], # 4
[30, 30, 0], # 5
[-45, 25, -25],
[-10, 30, -30], # 7
[45, 45, 45],
[-30, -30, 0],
[-45, 25, -25],
[30, 0, 45],
[15, -15, 25],
[10,-25, -15],
[-35, -15, 5],
[10, -30, 10],
[-25, 10, -5],
[-30, 5, 30],
[-30, 45, 30],#18
```

```
    [-15, 10, 15],
    [-5, -10, 5],
    [-10, 30, 40],
    [-30, -30, -30],]
    a = [-30, 45, 30]
    a = np.deg2rad(a)
    print(np.round(forward_kinematics(a[0],a[1],a[2])[0]))


    # rangle = []
    # for angle in degree_angles:
    #     rangle.append([np.deg2rad(angle[0]), np.deg2rad(angle[1]), np.deg2rad(angle[2])])

    # for angulo in rangle:
    #     # print(angulo)
    #     print(np.round(forward_kinematics(angulo[0],angulo[1],angulo[2])[0]))

    # [   0.   -0. -541.]
    # [   0.  225. -559.]
    # [-235.  136. -613.]
    # [ 245.  453. -572.]
    # [-223.  129. -872.]
    # [-374.  348. -690.]
    # [   0.    0. -993.]
    # [    0.    -0. -1106.]
    # [ 149.  -86. -620.]
    # [-303. -266. -559.]
    # [ 349.   52. -878.]
    # [ 269.   73. -772.]
    # [  56.  195. -669.]
    # [ 105. -184. -636.]
    # [ 237.  137. -687.]
    # [ -88. -185. -689.]
    # [ 160. -333. -681.]
    # [ 251.  -56. -942.]
    # [  33. -205. -757.]
    # [  92.  -18. -733.]
    # [  82. -371. -821.]

    angles = np.arange(-1.1, 1.1, 0.01).tolist()
    np.rad2deg(angles)

    # degree_angles = [
    # [-45, -45, -45],
    # [0, -45, -45],
    # [0, 0, -45],
    # [45, -45, 0],
    # [30, 30, 0],
    # [45, 30, -30],
    # [30, 30, 30],
    # [45, 45, 45],
    # [-30, -30, 0],
    # [-45, 30, -30],
    # [30, 0, 45],
    # [15, -15, 25],
    # [10,-25, -15],
    # [-35, -15, 5],
    # [10, -30, 10],
    # [-25, 10, -5],
    # [-30, 5, 30],
    # [25, 15, 45],
    # [-15, 10, 15],
    # [-5, -10, 5]]

    # len(degree_angles)

    import pandas as pd
    file_path                                                                    =
"/drive/MyDrive/TCC_JULHO_DE_2022/1000_EPISODES_FIXED_GOALS_TESTS/TESTS_FIXED_GOALS_WITH_DIS
TANCE_DATA/PONTO_10_/DISTANCES/REWARDS"
```

```
    file_name                           =                           "/REWARD_DATA_DIR_2022-09-
10_LR_=_0.0001_EPS_DECAY_=_0.999_EPS_MIN_=_0.01_N_EPISODES_=_1000_REWARD_DATA.txt"
    final_path = file_path + file_name
    # f = open(path, "r")

    # print(f.readlines())
    # import pandas as pd

    #     file_path     =     DRIVE_PATH     +     "/REWARDS/"+     "REWARD_DATA_DIR_2022-07-
31_LR_=_0.0001_EPS_DECAY_=_0.999_EPS_MIN_=_0.001_N_EPISODES_=_6904_REWARD_DATA.txt"
    #         print(DRIVE_PATH         +         "/REWARDS/"+         "REWARD_DATA_DIR_2022-07-
31_LR_=_0.0001_EPS_DECAY_=_0.999_EPS_MIN_=_0.001_N_EPISODES_=_6904_REWARD_DATA")

    file1 = pd.read_csv(final_path)
    file1.to_numpy
    count = 0

    numpyfile1 = file1.to_numpy()
    print(numpyfile1)
    for reward in numpyfile1:
        # print(reward)
        if(reward[2] == 10):
            count+=1
    count

    degree_angles = [
    [-45, -45, -45],
    [0, -45, -45],
    [0, 0, -45],
    [45, -45, 0],
    [30, 30, 0],
    [45, 30, -30],
    [30, 30, 30],
    [45, 45, 45],
    [-30, -30, 0],
    [-45, 30, -30],
    [30, 0, 45],
    [15, -15, 25],
    [10,-25, -15],
    [-35, -15, 5],
    [10, -30, 10],
    [-25, 10, -5],
    [-30, 5, 30],
    [25, 15, 45],
    [-15, 10, 15],
    [-5, -10, 5],
    [-10, 30, 40],
    [-30, -30, -30],]

    radian_angles = []
    rangle = []
    c = 0

    for angle in degree_angles:
        rangle.append([np.deg2rad(angle[0]), np.deg2rad(angle[1]), np.deg2rad(angle[2])])
    for line in rangle:
        c+=1
        print(c , line)

    # Commented out IPython magic to ensure Python compatibility.
    # %pylab inline
    import matplotlib.pyplot as plt
    import matplotlib.image as mpimg

    file_path     =     "/drive/MyDrive/TCC_JULHO_DE_2022/TESTS/RENDERS/RENDERS_DIR_2022-07-
24_LR_=_0.0001_EPS_DECAY_=_0.999_EPS_MIN_=_0.001_N_EPISODES_=_6904/step_1.png"

    img = mpimg.imread(file_path)
    imgplot = plt.imshow(img)
    plt.show()
```