

UNIVERSIDADE FEDERAL DO RIO GRANDE - FURG
CENTRO DE CIÊNCIAS COMPUTACIONAIS
CURSO DE ENGENHARIA DE COMPUTAÇÃO

Dissertação de Mestrado

**Aprendizado por Reforço Profundo para Navegação Sem
Mapa de um Veículo Híbrido Aéreo-Aquático usando
Imagens**

Junior Costa de Jesus

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal do Rio Grande - FURG, como requisito parcial para a obtenção do grau de Mestre em Engenharia de Computação

Orientador: Prof. Dr. Paulo Jorge Lilles Drews Jr
Co-orientador: Prof. Dr. Rodrigo da Silva Guerra

Rio Grande, 2023

Ficha Catalográfica

J58a Jesus, Junior Costa de.
Aprendizado por reforço profundo para navegação sem mapa de um veículo híbrido aéreo-aquático usando imagens / Junior Costa de Jesus. – 2023.
132 f.

Dissertação (mestrado) – Universidade Federal do Rio Grande – FURG, Programa de Pós-Graduação em Computação, Rio Grande/RS, 2023.
Orientador: Dr. Paulo Jorge Lilles Drews Jr.
Coorientador: Dr. Rodrigo da Silva Guerra.

1. Aprendizado por reforço profundo 2. Estimativa do mapa de profundidade 3. Robo autônomo 4. Veículo híbrido I. Drews Jr, Paulo Jorge Lilles II. Guerra, Rodrigo da Silva III. Título.

CDU 004

Catálogo na Fonte: Bibliotecário José Paulo dos Santos CRB 10/2344

DISSERTAÇÃO DE MESTRADO

**Aprendizado por Reforço Profundo para Navegação Sem Mapa de um
Veículo Híbrido Aéreo-Aquático usando Imagens**

Junior Costa de Jesus

Banca examinadora:

 Documento assinado digitalmente
ARMANDO ALVES NETO
Data: 03/02/2023 13:47:03-0300
Verifique em <https://verificador.iti.br>

Prof. Dr. Armando Alves Neto (UFMG) Documento assinado digitalmente
DANIEL FERNANDO TELLO GAMARRA
Data: 03/02/2023 17:07:45-0300
Verifique em <https://verificador.iti.br>

Prof. Dr. Daniel Fernando Tello Gamarra (UFSM) Documento assinado digitalmente
SILVIA SILVA DA COSTA BOTELHO
Data: 04/02/2023 08:06:09-0300
Verifique em <https://verificador.iti.br>

Profa. Dra. Silvia Silva da Costa Botelho (FURG) Documento assinado digitalmente
PAULO LILLES JORGE DREWS JUNIOR
Data: 03/02/2023 10:43:09-0300
Verifique em <https://verificador.iti.br>

Prof. Dr. Paulo Lilles Jorge Drews Junior (FURG)
Orientador

AGRADECIMENTOS

A concretização deste trabalho ocorreu, principalmente, pelo auxílio, compreensão e dedicação de várias pessoas. Agradeço a todos que, de alguma forma, contribuíram para conclusão deste estudo e, de uma maneira especial, agradeço:

- ao meu orientador Paulo Drews-Jr, por ter me guiado e aconselhado sobre os passos principais a serem dados para o meu projeto. Além disso, por ter me auxiliado durante a pandemia de 2020, um momento desafiador para todos nós.

- ao meu co-orientador Rodrigo Guerra, por ter me ajudado nesse projeto também.

- ao meu ex-orientador Daniel Gamarra, que foi quem me mostrou o quanto a área de pesquisa e robótica são fascinantes.

- à minha Mãe, Dulcilei, pelo apoio que me deu nessa jornada da minha vida, tanto financeiramente como mentalmente.

- aos meus amigos Marcelo “DJ” Back e Lucas Back, por me acompanharem desde Sapezal até Santa Maria. Neles encontrei muito do apoio nesse período de 2020.

- aos meus amigos Juliana “Juh” Adams e Hericles Farias, que sempre me ajudam quando mais preciso.

- ao meu amigo Jair “Jaja” Bottega, que jogamos muito vôlei no estilo Haikyuu.

- aos meus amigos Diego Cavalcante e Thiago “AxsHuHu” Cavalcante, que temos altas conversas fodas.

- ao meu amigo Alisson Kolling, que ainda não admitiu ter comido a pizza de frango do Kich

- ao meus amigos Victor Kich e Ricardo Grando, que viajaram comigo pelo Japão.

- à todos que contribuíram ou ajudaram no grupo VersusAI.

- à gurizada do Hydrone, por conversas semanais que me agregaram muito.

RESUMO

DE JESUS, Junior Costa. **Aprendizado por Reforço Profundo para Navegação Sem Mapa de um Veículo Híbrido Aéreo-Aquático usando Imagens.** 2023. 132 f. Dissertação (Mestrado) – Engenharia de Computação. Universidade Federal do Rio Grande - FURG, Rio Grande.

O Aprendizado por Reforço (RL) tem se mostrado altamente eficaz em jogos eletrônicos que utilizam imagens brutas de pixels e tarefas de controle contínuo. Entretanto, RL apresenta dificuldades em lidar com observações de alta dimensionalidade, como imagens brutas de pixels. É amplamente aceito que políticas de RL baseadas em estado físico, como medições de sensores a laser, geralmente produzem amostragens mais eficientes do que o aprendizado com base em pixels. Neste trabalho, é proposta uma nova abordagem que combina informações de uma estimativa de mapa de profundidade e imagens brutas de pixels para ensinar um agente de RL a realizar a navegação sem mapa em um Veículo Híbrido Aéreo-Aquático (HUAUV). Esta abordagem, denominada Representações Priorizadas Contrastivas Não Supervisionadas de Imagens de Profundidade e Pixel em Aprendizado por Reforço (CUPRL e Depth-CUPRL), estima a profundidade de imagens e utiliza imagens brutas de pixels com uma memória de repetição priorizada. É utilizada uma combinação de RL e Aprendizagem Contrastiva para lidar com o desafio de aprender com base em observações de imagens. A Aprendizagem Contrastiva permite criar um espaço latente que é capaz de mapear imagens de pixel e profundidade de tal forma que, mesmo ao utilizar apenas imagens de pixels, é possível criar representações eficientes para solucionar problemas de navegação em ambientes complexos. Os resultados obtidos com o HUAUV indicam que a abordagem proposta é eficaz na tomada de decisão e supera as abordagens baseadas em pixels existentes na capacidade de navegação sem mapa em um contexto 2D e 3D.

Palavras-chave: Aprendizado por Reforço Profundo, Estimativa do mapa de profundidade, Robô Autônomo, Veículo Híbrido.

ABSTRACT

DE JESUS, Junior Costa. **Deep Reinforcement Learning for Mapless Navigation of an Hybrid Unmanned Aerial-Underwater Vehicle using Images**. 2023. 132 f. Dissertação (Mestrado) – Engenharia de Computação. Universidade Federal do Rio Grande - FURG, Rio Grande.

Reinforcement Learning (RL) has shown impressive performance in video games through raw pixel images and continuous control tasks. However, RL has poor performance with high-dimensional observations such as raw pixel images. It is generally accepted that RL policies based on physical state, such as laser sensor measurements, provide more efficient sampling results than pixel-based learning. This work presents a new approach that extracts information from a depth map estimate and raw pixel images to train an RL agent to perform map-less navigation of a Hybrid Unmanned Aerial-Underwater Vehicle (HUAUV). The proposed approach is called Unsupervised Prioritized Contrastive Representations of Depth and Pixel Images in Reinforcement Learning (CUPRL and Depth-CUPRL) which estimates depth from images and uses raw pixel images with a prioritized replay memory. A combination of RL and Contrastive Learning is used to address the problem of RL based on image observations. Contrastive Learning allows to create a latent space that is capable of mapping pixel and depth images in a way that, even using only pixel images, it is possible to create efficient representations to solve navigation problems in complex environments. From the results obtained with the HUAUV, it can be concluded that the proposed CUPRL and Depth-CUPRL approach is effective for decision making and outperforms state-of-the-art pixel-based approaches in map-less navigation in a 2D and 3D context.

Keywords: Deep-RL, Depth Map Estimation, Autonomous Robot, Hybrid Vehicle.

LISTA DE FIGURAS

1	Interação do agente no ambiente em aprendizado por reforço	17
2	Arquitetura Ator-Crítica	22
3	Sistema da CUPRL	24
4	Sistema da Depth-CUPRL	25
5	Estrutura de aprendizagem CURL	27
6	Arquitetura para profundidade e pose CNN	28
7	Arquitetura da rede CUPRL e Depth-CUPRL	33
8	Ambiente de treinamento usado na Simulação do Gazebo.	37
9	Ambiente de avaliação usado na Simulação do Gazebo	38
10	Veículo simulado baseado no veículo real Hydrone.	38
11	Recompensas do primeiro ambiente simulado - Contexto 2D	40
12	Recompensas do segundo ambiente simulado - Contexto 2D	41
13	Recompensas do primeiro ambiente simulado - Contexto 3D	43
14	Como ocorre avaliação das redes contrastivas	45
15	Caminho durante avaliação das técnicas no primeiro ambiente	45
16	Imagem da água em pixel e mapa de profundidade, onde imagens da interface ar-água e água-ar são vistas como obstáculos nas imagens de profundidade	47
17	Recompensas do segundo ambiente simulado - Contexto 3D	48
18	Caminho durante avaliação das técnicas no segundo ambiente	49

LISTA DE TABELAS

1	Especificação feita no Hydrone	39
2	Comparação entre proposta e outras redes no primeiro ambiente - Contexto 2D	40
3	Comparação entre proposta e outras redes no segundo ambiente - Contexto 2D	42
4	Posição inicial do veículo e ponto alvo na avaliação	44
5	Testes no primeiro ambiente de avaliação - Contexto 3D	44
6	Testes no segundo ambiente de avaliação - Contexto 3D	49

LISTA DE ABREVIATURAS E SIGLAS

UAV	Veículo Aéreo não Tripulado
RL	Aprendizado por Reforço
Deep-RL	Aprendizado por Reforço Profundo
DQN	Deep Q-Network
SAC	<i>Soft Actor-Critic</i>
ROS	<i>Robot Operating System</i>
UUV	Veículo Subaquático não Tripulado
UV	Veículo não Tripulado
UGV	Veículo Terrestre não Tripulado
CUPRL	Representações Priorizadas Contrastivas Não Supervisionadas de Imagens de Profundidade e Imagens Brutas de Pixel em Aprendizado por Reforço
Depth-CUPRL	Representações Priorizadas Contrastivas Não Supervisionadas de Imagens de Profundidade em Aprendizado por Reforço

SUMÁRIO

1	INTRODUÇÃO	11
1.1	Problema	13
1.2	Objetivo Geral	13
1.3	Objetivos Específicos	13
1.4	Contribuição Científicas	14
1.5	Seções do trabalho	14
2	REVISÃO DA LITERATURA	15
2.1	Trabalhos Relacionados	15
2.2	Fundamentação Teórica	17
2.2.1	Aprendizado por Reforço	17
3	METODOLOGIA	23
3.1	Aprendizado por Reforço Profundo	24
3.1.1	Soft Actor-Critic	24
3.1.2	Representações Contrastivas Não Supervisionadas para Aprendizagem por Reforço	26
3.2	Estimativa de profundidade	27
3.3	Repetição de Experiência Priorizadas	29
3.4	Estrutura de rede CUPRL e Depth-CUPRL	29
3.4.1	Função de recompensa	30
4	RESULTADOS EXPERIMENTAIS	35
4.1	Configuração do Experimento	36
4.1.1	ROS e Gazebo	36
4.1.2	Ambientes de Simulação	36
4.1.3	Hydrone	37
4.2	Ambiente Aéreo Contexto 2D	39
4.3	Ambiente Aéreo-aquático Contexto 3D	42
5	CONSIDERAÇÕES FINAIS E PERSPECTIVAS	52
	REFERÊNCIAS	54
	APÊNDICE A CUPRL	59
	APÊNDICE B DEPTH-CUPRL	97

1 INTRODUÇÃO

Os veículos não tripulados, conhecidos como UVs (*Unmanned Vehicles*, em inglês), são uma categoria de robôs que desempenham papéis importantes em uma variedade de missões aéreas e aquáticas, incluindo ciências marinhas, extração de petróleo e gás, inspeção, exploração e resgate [Cho et al., 2015]. Esses tipos de robôs podem ser divididos em três categorias: terrestres, aéreos e aquáticos. Esses veículos são capazes de transportar instrumentos, coletar amostras e conduzir pesquisas, permitindo que os cientistas acompanhem o progresso de suas missões de forma segura a bordo de um navio.

A principal diferença entre esses robôs é o meio em que realizam suas missões. Para os UVs terrestres ou UGVs (*Unmanned Ground Vehicles*, em inglês). No ambiente aéreo, os veículos mais comuns são os drones, classificados como veículos aéreos não tripulados ou UAVs (*Unmanned Aerial Vehicles*, em inglês). Na categoria subaquática, tem-se os veículos subaquáticos não tripulados ou UUVs (*Unmanned Underwater Vehicles*, em inglês). Na indústria do petróleo, esses robôs podem ser utilizados como olhos dos operadores subaquáticos e são equipamentos fundamentais durante as instalações. Além disso, os UVs podem realizar inspeções visuais de alta resolução em estruturas submarinas tridimensionais para exploração de petróleo e gás, como dutos, cascos de navios e plataformas.

O meio em que um UV atua define sua categoria. No entanto, em algumas situações pode ser necessário atuar ou detectar em mais de um meio. Por exemplo, a inspeção de uma plataforma de petróleo precisa ser realizada tanto na parte submersa quanto na parte aérea. Para ambos os cenários, UAVs ou UUVs não são capazes de desempenhar de forma satisfatória para um monitoramento completo [Sudevan et al., 2018]. Portanto, tem-se uma crescente necessidade no desenvolvimento de veículos híbridos não tripulados (HUVs) que unam e explorem as vantagens e desvantagens dos UAVs, UUVs e UGVs [Drews et al., 2014]. Especificamente, o desenvolvimento de veículos híbridos capazes de operar tanto em ambientes aéreos quanto subaquáticos, chamados de HUAUVs (*Hybrid Unmanned Aerial Underwater Vehicle*, em inglês), tem sido um grande desafio. Esses modelos de veículos requerem técnicas que consigam lidar com os contrastes entre os ambientes da água e do ar tanto na percepção quanto na atuação. Vários modelos de HU-

AUVs já foram propostos tanto na literatura [Zeng et al., 2022]. Algumas das primeiras propostas foram feitas por Drews et al. [2014], Yang et al. [2015] e Maia et al. [2015]. Um veículo híbrido deste tipo permite realizar diversas tarefas, como inspeção e manutenção, mapeamento e coleta de dados, além de vigilância, busca e salvamento.

O problema da navegação robótica para veículos UAVs, UUVs e UGVs autônomos, também conhecidos como HUAUVs, pode ser definido como a necessidade de perceber o ambiente e planejar movimentos que evitem riscos [Alves et al., 2011]. Isso pode ser alcançado através do uso de sensores, como *lasers*, câmeras e sensores de profundidade, para perceber o ambiente e sistemas de planejamento de trajetórias para evitar situações perigosas. No entanto, a navegação em meios contrastantes pode ser desafiadora para esses veículos. Para lidar com esse problema, Grando [2021] propôs um sistema de planejamento inteligente baseado em aprendizado por reforço com o uso de percepções a *laser*.

O problema de navegação em HUAUVs pode ser abordado através do uso do aprendizado por reforço [Jesus et al., 2019] e [Grando et al., 2020]. Esta técnica de aprendizado de máquina se concentra na tomada de decisão sequencial e no aprimoramento gradual do comportamento do agente através da experiência de tentativa e erro. Um dos aspectos-chave do aprendizado por reforço é que o agente não precisa de conhecimento completo do ambiente, mas sim de capacidade de interagir e coletar informações [François-Lavet et al., 2018]. O agente realiza ações e é recompensado ou punido de acordo com o novo estado resultante, criando assim um ciclo entre ações e interpretações do ambiente.

Com este contexto, o presente trabalho propõe um sistema de navegação para um HUAUV, que seja capaz de desenvolver uma estratégia e política a partir de características extraídas de imagens de pixel e de profundidade. É proposta a técnica chamada de Representações Priorizadas Contrastivas Não Supervisionadas de Imagens de Profundidade e Imagens Brutas de Pixel em Aprendizado por Reforço (CUPRL), que considera informações da profundidade de imagens e imagens brutas de pixel, e Representações Priorizadas Contrastivas Não Supervisionadas de Imagens de Profundidade em Aprendizado por Reforço (Depth-CUPRL), que considera somente informações da profundidade de imagens com uma memória de repetição priorizada para realizar a navegação de um HUAUV. Além disso, verifica-se que o uso de uma combinação de Deep-RL e Aprendizagem Contrastiva treinada com imagens de profundidade e imagens brutas, leva a um desempenho superior quando comparado com outras abordagens de Deep-RL baseadas apenas em imagens brutas, mesmo quando aplicadas apenas as imagens brutas depois do treinamento. Com HUAUVs, seria possível realizar inspeções em ambientes de difícil acesso que precisam lidar com navegação tanto na água quanto no ar.

O presente trabalho propõe um sistema de navegação para HUAUVs, utilizando técnicas de Aprendizado por Reforço Profundo (Deep-RL) para fornecer uma resposta de ação eficiente em ambientes reais. Dado o histórico da FURG com o desenvolvimento

robótico dos HUVs [Drews et al., 2014], [Horn et al., 2019] e [Grando et al., 2021a], o projeto é viável para ser realizado em HUAUVs. No entanto, uma das principais limitações das técnicas de Aprendizado por Reforço é a necessidade de grandes quantidades de dados difíceis de serem coletadas no mundo real [Bonatti et al., 2019]. Além disso, Aprendizado por Reforço tem mostrado ser ineficiente com observações de alta dimensão, como imagens de pixel bruto [Lake et al., 2017]. Considerando essas limitações, o presente trabalho propõe o uso de técnicas que extraíam informações mais úteis de imagens, como mapas de profundidade em combinação com imagens brutas de pixel, para a percepção do ambiente.

A utilização de técnicas de aprendizado por reforço com base em mapas de profundidade pode melhorar significativamente o desempenho de redes Deep-RL ao controlar robôs aéreo-aquáticos, além de fornecer uma dinâmica de navegação eficiente em ambientes híbridos. Os HUAUVs podem realizar operações contínuas sem intervenção humana, tanto na água quanto no ar, e podem trazer vantagens significativas para a indústria de gás e energia *offshore*. Isso os torna ideais para inspeções em locais de difícil acesso. Porém, é crucial que os agentes de aprendizado por reforço possam tomar decisões rápidas e desenvolver estratégias para lidar com ambientes complexos.

1.1 Problema

O objetivo deste trabalho é investigar o problema da ineficiência das redes Deep-RL ao lidar com imagens [Lake et al., 2017] e expandir o escopo para a navegação de Veículos Híbridos Aéreo-Aquáticos (HUAUVs). Embora a navegação em HUAUVs com Deep-RL tenha sido mostrada eficiente ao lidar com dados de sensores [Grando et al., 2022], ainda não há trabalhos que abordem essa questão ao utilizar imagens para controlar veículos híbridos em ambientes aéreo e subaquático.

1.2 Objetivo Geral

O presente trabalho propõe o uso de abordagens baseadas em Deep-RL combinadas com imagens e informação de profundidade para a navegação sem mapa orientada a ponto alvo em ambientes híbridos (água-ar e ar-água) usando um HUAUV em cenários simulados. Isso permitirá uma melhor estimativa do ambiente e uma resposta de ação mais precisa para realizar tarefas de navegação de forma eficiente e segura.

1.3 Objetivos Específicos

- Analisar a capacidade de evitar colisão com obstáculos em cenários específicos.
- Fazer uma análise comparativa entre as abordagens de Deep-RL propostas ao problema.

- Comparar a navegação sem mapa realizada pelos agentes de Deep-RL com a navegação realizada por um controlador de rastreamento geométrico tradicional.
- Navegar sem um mapa aéreo ou subaquático em um HUAUV em ambiente 2D, utilizando apenas informações de localização e visão obtidas pela câmera.
- Navegar sem um mapa aéreo ou subaquático para o HUAUV em ambiente 3D, utilizando apenas informações de localização e visão obtidas pela câmera.

1.4 Contribuição Científicas

Até o presente momento, este trabalho já publicou os resultados da navegação em contexto 2D, que introduzem o conceito da Depth-CUPRL na Conferência Internacional de Robótica e Sistemas Inteligentes (IEEE IROS 2022) [de Jesus et al., 2022] e contribuiu para os trabalhos de Jesus et al. [2021], Grando et al. [2021b] e Grando et al. [2022].

1.5 Seções do trabalho

A divisão do trabalho foi feita da seguinte forma: Na sequência, o Capítulo 2 apresenta uma revisão do estado da arte sobre trabalhos pertinentes a este trabalho. Após isso, no Capítulo 3 é apresentada a metodologia proposta, bem como uma discussão de ferramentas utilizadas, adaptadas ou desenvolvidas. Os resultados preliminares obtidos são apresentados no Capítulo 4. Por último, no Capítulo 5 as considerações finais e perspectivas para futuras pesquisas.

2 REVISÃO DA LITERATURA

O estudo de Drews et al. [2014] apresenta uma revisão histórica sobre o desenvolvimento de HUAUVs, desde as primeiras ideias e protótipos até a data de publicação do trabalho. Além disso, o estudo também realiza uma análise comparativa entre os diferentes tipos de veículos aéreos e subaquáticos propostos e/ou desenvolvidos, destacando as vantagens e desvantagens de cada modelo. Conclui-se que um veículo baseado em uma estrutura de quadrotor é a opção mais adequada para o desenvolvimento de um HUAUV.

Este trabalho proposto por Drews et al. [2014] é um dos primeiros a abordar o tema de HUAUVs. Atualmente, ainda há poucos estudos desenvolvidos nessa área, com a maioria deles se concentrando nas questões mecânicas e estruturais do veículo, e poucos avanços no que diz respeito à inteligência do veículo. Isso é um dos principais desafios para o desenvolvimento de estudos sobre navegação para HUAUVs, tanto com mapa quanto sem mapa, pois o enfoque tem sido na física do objeto em vez de no planejamento de movimentação. Portanto, o problema da navegação sem mapa, amplamente estudado em robôs móveis terrestres, ainda precisa ser investigado para HUAUVs.

Este trabalho propõe abordar o desafio da navegação sem mapa em HUAUVs, utilizando técnicas baseadas em informações visuais e de profundidade para melhorar a precisão e eficiência do veículo. Neste capítulo, serão apresentadas as principais publicações relacionadas a esse assunto, bem como uma revisão das técnicas teóricas que serão aplicadas para resolver o problema proposto. É importante destacar que, embora haja algumas pesquisas sobre HUAUVs, ainda há pouco avanço em termos de inteligência do veículo e, portanto, a navegação sem mapa é uma área ainda pouco explorada.

2.1 Trabalhos Relacionados

O estudo da navegação sem mapa tem sido amplamente pesquisado para robôs móveis terrestres [Tai and Liu, 2016]. No entanto, a navegação autônoma de robôs móveis aéreos usando abordagens de Deep-RL é menos comum e geralmente se concentra em evitar o uso de informações visuais [Grando et al., 2022, 2020] ou em utilizar informações simplificadas sem o recurso de aprendizado contrastivo [Rodriguez-Ramos et al., 2018,

Sampedro et al., 2019, He et al., 2020, Li et al., 2020].

Rodriguez-Ramos et al. [2018] propõe uma abordagem baseada em Deep-RL para resolver o problema de pouso de um UAV em uma base que pode estar em movimento ou estática. Sampedro et al. [2019] sugere que uma abordagem baseada em uma técnica Deep-RL poderia realizar uma tarefa de Busca e Resgate em cenários fechados, utilizando o algoritmo Deep Deterministic Policy Gradient (DDPG) [Lillicrap et al., 2015]. No trabalho de Jesus et al. [2021], é demonstrado que abordagens baseadas no algoritmo Soft Actor-Critic (SAC) [Haarnoja et al., 2018b] são mais eficazes para a navegação de robôs do que abordagens baseadas no algoritmo DDPG.

O uso de técnicas de aprendizado profundo como o Deep-RL para a navegação de sistemas robóticos, como HUAUVs, pode ser desafiador devido à necessidade de grandes quantidades de dados para treinamento e a ineficiência de observações de alta dimensão, como imagens de pixel bruto [Kaiser et al., 2019]. Para superar esses desafios, Laskin et al. [2020] propôs a técnica CURL, que é capaz de extrair características úteis das imagens brutas e melhorar o desempenho do controle de redes Deep-RL. Embora a CURL tenha mostrado resultados promissores em ambientes simulados semelhantes a jogos, este projeto busca adaptá-la para ambientes mais próximos das condições reais, com visão de primeira pessoa e usando imagens monoculares.

Para navegação de veículos aéreos móveis, é possível encontrar trabalhos como o de Thomas et al. [2021] que apresenta um algoritmo baseado em RL utilizando modelos de auto-atenção para controlar um UAV autônomo. O trabalho de Thomas et al. [2021] mostrou que ele pode efetivamente completar a navegação do veículo mesmo quando submetido a diferentes entradas no algoritmo, destacando como o algoritmo é capaz de lidar com dados de estado com ruídos ou modificações. Outro trabalho, He et al. [2020], usou um Lobula Giant Moment Detector (LGMD) para simplificar as informações de visão para Deep-RL na navegação e prevenção de obstáculos de um UAV. Ele realizou missões em um ambiente complexo com 80% de taxa de sucesso.

Este trabalho se destaca de outros estudos relacionados ao utilizar informações de mapas de profundidade em uma abordagem baseada em aprendizado contrastivo para lidar com o desafio das observações de alta dimensão. Ele também propõe um sistema de memória de repetição de experiências priorizado que aumenta a eficiência da abordagem proposta, e é o primeiro a propor um sistema de aprendizado contrastivo com repetição de experiências priorizadas. A utilização do aprendizado contrastivo neste trabalho permite desenvolver um espaço latente que pode relacionar imagens de pixels e profundidade de forma que, mesmo usando somente imagens de pixels, é possível criar representações eficientes para resolver problemas de navegação em ambientes complexos em contexto 2D e 3D.

2.2 Fundamentação Teórica

2.2.1 Aprendizado por Reforço

O aprendizado por reforço é uma técnica de aprendizado de máquina que se concentra em como um agente pode aprender a tomar decisões sequenciais em um ambiente. O método adotado neste trabalho, baseado em três aspectos principais [François-Lavet et al., 2018], consiste em permitir que o agente aprenda um comportamento ótimo por meio da interação com o ambiente. Ele utiliza a experiência adquirida por tentativa e erro, sem exigir conhecimento completo do ambiente, mas apenas a capacidade de interagir e coletar informações. O agente realiza ações no ambiente e recebe uma recompensa pelas decisões de ação tomadas, criando um ciclo entre ações e interpretações, como ilustrado na Figura 1. A figura ilustra como a função de recompensa, representada pelo “interpretador”, determina a otimização das ações e interações no ambiente.

Figura 1: Interação do agente no ambiente em aprendizado por reforço



Fonte: Super Data Science

O aprendizado por reforço se divide em duas configurações, a *offline* e a *online*, como é mencionado na literatura [Sutton et al., 1998]. Na configuração *offline*, a experiência é coletada previamente e então utilizada para o aprendizado. Isso é diferente da configuração *online* onde os dados são fornecidos de forma sequencial e usados para atualizar progressivamente o comportamento do agente.

2.2.1.1 Processo de decisão de Markov

A forma básica de modelar o aprendizado por reforço é seguindo um processo de Markov. O processo de Markov é definido como a probabilidade do estado futuro depender apenas do estado e da ação atual [Bellman, 1957]. Isso quer dizer:

$$P(s_{t+1} \mid s_0, a_0, s_1, a_1, \dots, s_t, a_t) = P(s_{t+1} \mid s_t, a_t) \quad (1)$$

onde $P(s_{t+1} \mid s_t, a_t)$ é a probabilidade de transição, no tempo de t , do estado s_t até o estado s_{t+1} sob a ação a_t .

O objetivo do aprendizado por reforço é fazer com que o agente maximize a recompensa que ele recebe ao longo do tempo, tomando ações baseadas no estado atual e no ambiente. Isso é feito por meio de uma função de recompensa, que define a meta a ser alcançada pelo agente. À medida que o agente realiza ações, ele passa por diferentes estados e é recompensado de acordo com a qualidade das ações tomadas.

Com a abordagem baseada no processo de Markov, o agente utiliza os seguintes passos em cada tempo t :

1. O agente observa o estado atual s_t ou s .
2. O agente faz a ação a_t ou a .
3. O agente recebe a recompensa $r_t = R(s_t, a_t)$, sendo $R(s_t, a_t)$ a recompensa imediata após a transição de s_t com a ação a_t .
4. O agente entra no próximo estado s_{t+1} ou s' .

A abordagem do processo de Markov utiliza a equação de Bellman para determinar qual é a ação ótima para um agente em um determinado estado, levando em conta o desconto temporal que penaliza políticas que são desnecessárias ou demoradas para alcançar o objetivo. Em outras palavras, a equação de Bellman prevê a recompensa gerada a partir de um estado para outro, enquanto a função de recompensa apenas considera o que é benéfico no curto prazo. A equação de Bellman é definida como:

$$V(s_t) = \max_{a_t} (R(s_t, a_t) + \gamma V(s_{t+1})) \quad (2)$$

Essa fórmula nos diz qual é o maior valor esperado do estado, onde V é definido como o “valor” do estado no ambiente. O valor γ (gama) é um número escalar entre zero e um utilizado para admitir um valor de desconto para a importância da futura recompensa. Com a equação de Bellman e as probabilidades de cada ação possível, é possível determinar a equação do processo de decisão de Markov, que é essencial para o aprendizado por reforço:

$$V(s_t) = \max_{a_t} (R(s_t, a_t) + \gamma \sum_{s_{t+1}} P(s_{t+1} \mid s_t, a_t) V(s_{t+1})) \quad (3)$$

Com a incorporação da incerteza nas ações, o processo de decisão de Markov permite modelar um ambiente de previsão mais preciso em um contexto mais abstrato e estocástico. Isso é diferente de uma abordagem anterior, onde todas as ações eram tomadas em um ambiente determinístico sem considerar a probabilidade de a ação ser realizada.

2.2.1.2 *Q-Learning*

Uma das diferenças entre um agente que toma decisões baseado no processo de decisão de Markov e o *Q-Learning* é que, na abordagem Markoviana, o agente avalia a qualidade dos estados futuros com base no estado atual, enquanto no *Q-Learning*, o agente avalia a qualidade de cada ação considerando o estado atual [Watkins and Dayan, 1992]. A função ‘*Q*’ é definida como a ”qualidade” da ação de um agente em um determinado estado. A equação do *Q-Learning* pode ser derivada a partir da abordagem de Markov, dando origem à seguinte equação:

$$Q(s_t, a_t) = R(s_t, a_t) + \gamma \sum_{s_{t+1}} P(s_{t+1} | s_t, a_t) \max_{a_{t+1}} (Q(s_{t+1}, a_{t+1})) \quad (4)$$

Mesmo após encontrar o valor de ‘*Q*’ para uma determinada ação em um estado, é importante lembrar que o agente precisará explorar esse ambiente provavelmente mais de uma vez. Para permitir o aprendizado por reforço, é necessário considerar uma diferença temporal e uma taxa de aprendizado (α) para que o algoritmo possa melhorar o seu desempenho ao longo do tempo, mesmo quando as ações não beneficiam o agente. Em outras palavras, é possível definir as seguintes equações:

1. A equação da diferença temporal entre os valores de *Q*:

$$TD_t = \Delta Q(s_t, a_t) \quad (5)$$

$$TD_t = R(s_t, a_t) + \gamma \sum_{s_{t+1}} P(s_{t+1} | s_t, a_t) (\max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q_{t-1}(s_t, a_t)) \quad (6)$$

2. A equação que atualizar o valor de *Q* para cada ação que passou por aquele estado:

$$Q_t(s_t, a_t) = Q_{t-1}(s_t, a_t) + \alpha TD_t(s_t, a_t) \quad (7)$$

Com a equação 7 definida, é possível construir um agente inteligente que aprende por meio de reforços e melhora seu desempenho com o passar do tempo.

2.2.1.3 *Política de Gradiente*

Uma política define como um agente seleciona ações. Elas podem ser classificadas como estacionárias ou não-estacionárias. Uma política não-estacionária muda com o

tempo e é útil para contextos limitados, onde a recompensa acumulativa que o agente busca otimizar é limitada a um número finito de intervalos de tempo.

Políticas também podem ser categorizadas sobre um segundo critério de serem determinísticas ou estocásticas:

- No caso de ser determinística, a política é descrita por $\mu(s)$.
- No caso de ser estocástica, a política é descrita por $\pi(s, a)$, onde se denota a probabilidade de que a ação a possa ser escolhida no estado s .

2.2.1.4 *Aprendizado de política-on e política-off*

Métodos baseados em política-on buscam avaliar e melhorar a política atualmente utilizada para tomar decisões, enquanto métodos baseados em política-off avaliam e melhoram uma política diferente daquela usada para gerar dados. Em métodos baseados em política-off, o aprendizado é mais direto quando se utilizam trajetórias que não foram obtidas necessariamente com base na política atual, mas sim com base em outra política de comportamento. Nestes casos, a repetição de experiências permite o uso de experiências passadas para aprendizado sem a introdução de tendências. Já os métodos baseados em política-on geralmente introduzem tendências quando utilizam memórias de repetição.

2.2.1.5 *Ator-Crítica*

A política pode ser representada por redes neurais que são atualizadas tanto de forma determinística quanto estocástica. Para ambos os casos, a política de gradiente necessita de uma estimativa da função de valor da política atual. Uma abordagem comum é usar uma arquitetura Ator-Crítica, mostrada na Figura 2, que consiste em duas partes: um ator e uma crítica [Konda and Tsitsiklis, 2000]. O ator se refere à política e a crítica é usada para estimar a função de valor, podendo ser $Q(s, a)$ ou $V(s)$. Em Deep-RL, tanto o ator quanto a crítica podem ser representados por aproximadores de função de rede neural não-linear. Em resumo, a rede ator é responsável pela exploração das ações do agente no ambiente, enquanto a rede crítica guia a rede ator informando se a ação é boa ou ruim. O algoritmo Ator-Crítica é descrito no Algoritmo 1.

Os conceitos teóricos aqui apresentados têm como objetivo esclarecer o que é o Aprendizado por Reforço e suas premissas fundamentais. O aprendizado por reforço se baseia em três conceitos-chave: o estado s_t , que descreve o ambiente no momento atual; a ação a_t , que é a ação tomada pelo agente no estado atual; e a recompensa r_t , que é a recompensa ou punição recebida pelo agente como resultado da ação tomada. O objetivo do agente é encontrar uma política de ação que maximiza a recompensa esperada ao longo do tempo, aprendendo com as ações e recompensas passadas. Além disso, o aprendizado por reforço também é conhecido por sua capacidade de lidar com problemas de tomada de decisão estocásticos e com incerteza, tornando-o uma ferramenta valiosa em várias áreas, como inteligência artificial, robótica e engenharia de sistemas.

Algoritmo 1: Arquitetura Ator-Crítica

Entradas: Parâmetros iniciais da rede ator ϕ , parâmetros da rede crítica θ , memória de repetição M , número máximo de intervalos de tempo T

Define os parâmetros das redes alvos $\theta' \leftarrow \theta$

para $t = 1$ **até** T **faça**

 Observa estado s e seleciona ação $a \sim \pi_\phi(\cdot|s)$ e Executa a no ambiente

 Observa próximo estado s' , e recompensa r

 Armazena (s, a, r, s') na memória M

se Atualiza rede RL **então**

 Seleciona aleatoriamente um *batch*, $B = \{(s, a, r, s')\}$ de M

 Computa a correção para o valor da ação Q:

$$y(r, s') = r + \gamma = Q_{\theta'}(s', a')$$

 Atualiza a rede crítica:

$$\nabla_{\theta} \frac{1}{|B|} \sum_{(s,a,r,s') \in B} (Q_{\theta}(s, a) - y(r, s'))^2$$

 Atualiza a política da rede ator:

$$\nabla_{\phi} \frac{1}{|B|} \sum_{s \in B} Q_{\theta_i}(s, a_{\phi})$$

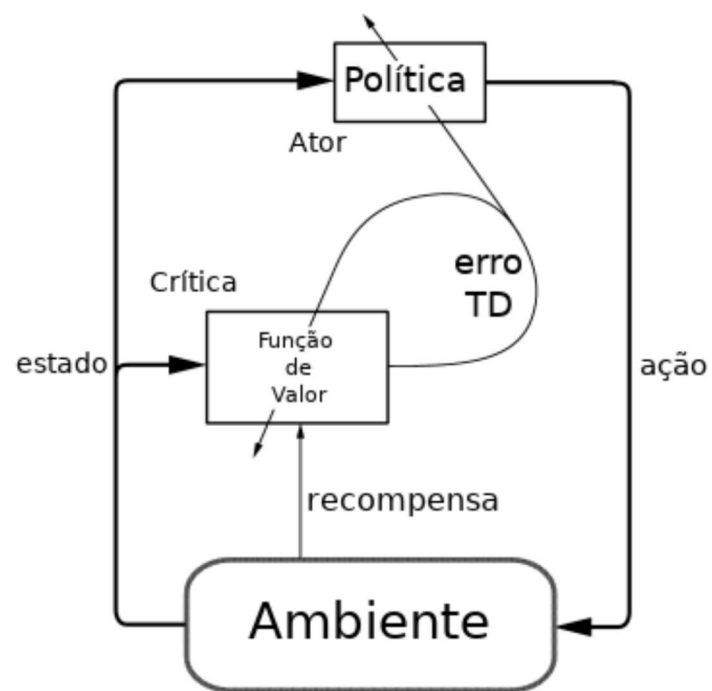
 Atualiza as redes alvo:

$$\theta' \leftarrow \tau\theta + (1 - \tau)\theta$$

fim

fim

Figura 2: Arquitetura Ator-Crítica



Fonte: Autor

3 METODOLOGIA

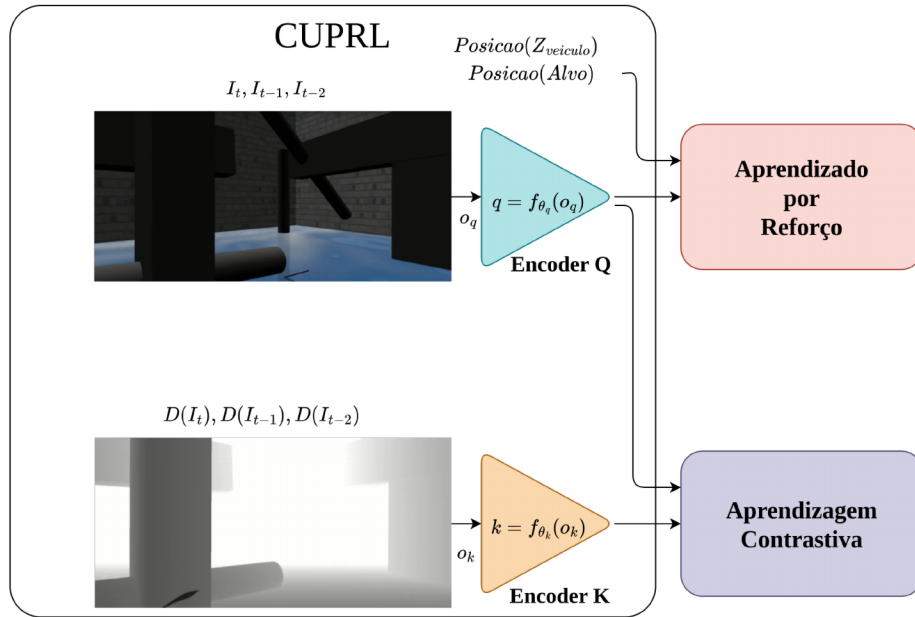
Este trabalho propõe um sistema de controle para HUAUV utilizando informações visuais e de profundidade como entrada para redes de aprendizado por reforço profundo (Deep-RL) para construir uma política de movimento que evita obstáculos no ambiente. São geradas imagens de profundidade e imagens brutas de pixels que são enviadas como entrada para a rede neural similar a proposta por [Laskin et al., 2020], chamada CURL, que aprende a controlar a navegação nestes ambientes através de recompensas priorizadas. Os métodos propostos neste trabalho, denominados CUPRL e Depth-CUPRL, são baseados em uma rede CURL combinada com mapas de profundidade, imagens brutas de pixels e memórias priorizadas para a navegação de um veículo híbrido e são originados pelo próprio autor do trabalho. A equação de movimento para CUPRL e Depth-CUPRL é definida como:

$$v_t = f^X(I_t), X \in \{CUPRL, Depth-CUPRL\} \quad (8)$$

onde I_t é a imagem de pixel bruta da câmera e v_t é a velocidade aplicada ao HUAUV. Para o treinamento do CUPRL, a Equação 8 recebe uma estimativa de profundidade de I_t e a informação da observação em pixel. A rede neural CUPRL extrai informações do mapa de profundidade e imagens de pixel para depois passar por uma rede SAC que retorna a velocidade ao robô. Na Figura 3 é mostrado o sistema de estados da CUPRL, I_t, I_{t-1}, I_{t-2} ou o_q e $D(I_t, I_{t-1}, I_{t-2})$ ou o_k que serão utilizados nas redes para o treinamento. $D(I_t, I_{t-1}, I_{t-2})$ é observação dos mapas de profundidades que foram gerados das imagens de pixel de I_t, I_{t-1}, I_{t-2} . Essas observações são passadas para seus codificadores f_{θ_q} e f_{θ_k} que retornam as representações de espaço latente q e k . O espaço latente q , a altura do veículo no eixo z e a posição do alvo no ambiente são as informações passada para o algoritmo Deep-RL. Já para a aprendizagem contrastiva, ambos espaços latentes q e k são utilizados no aprendizado das características de o_q e o_k .

A rede neural Depth-CUPRL, diferentemente da CUPRL, se concentra em extrair informações apenas a partir dos mapas de profundidade, como pode ser observado na Figura 4. As diferenças entre as duas redes propostas estão no tipo de informação que elas priorizam. No caso da Depth-CUPRL, as redes neurais recebem informações apenas

Figura 3: Sistema da CUPRL



Fonte: Autor

de mapas de profundidades $D(I_t, I_{t-1}, I_{t-2})$ com modificações nessas observações para o treinamento. Por usar um contexto de modificações de recorte nas observações o_q e o_k , a Depth-CUPRL pode convergir mais rápido no contexto de aprender as características importantes do ambiente. Já o caso da CUPRL, uma observação representa informações visuais e outra observações informações de profundidade. O caso da CUPRL toma mais tempo para aprender as características do ambiente.

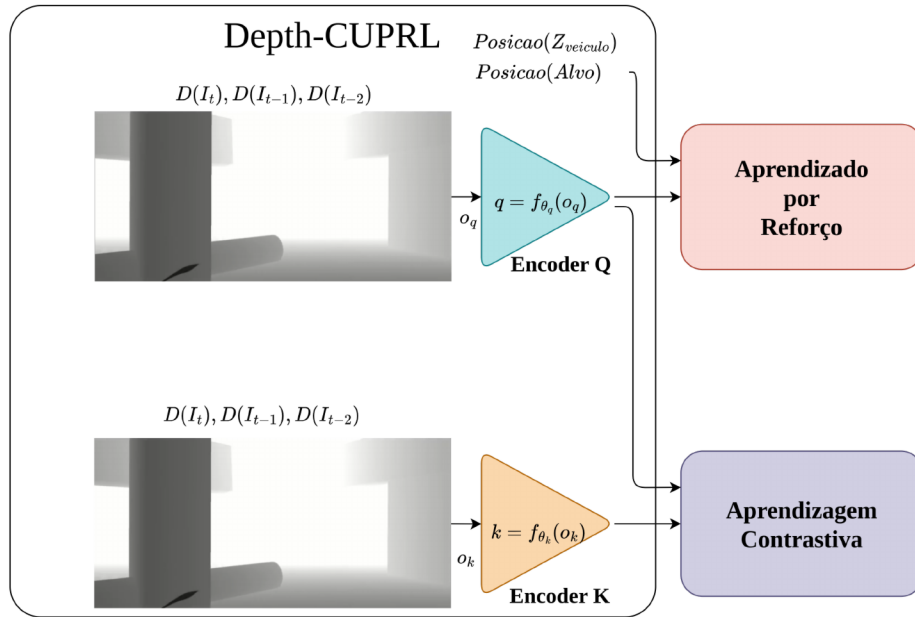
3.1 Aprendizado por Reforço Profundo

O problema de aprendizado por reforço profundo é basicamente o controle de um agente em um ambiente para maximizar sua função de recompensa. O algoritmo *deep Q-network* (DQN) proposto em [Mnih et al., 2013], [Mnih et al., 2015] foi eficaz em muitos jogos Atari, conseguindo atingir um desempenho humano usando apenas uma entrada de imagem de pixel bruto para estimar a ação do agente. No entanto, o DQN é limitado ao lidar com espaços de ação discretos, o que torna ineficaz para tarefas de robótica que exigem espaços de ação contínuos. Portanto, é necessário aplicar outros algoritmos para lidar com essas tarefas de domínio contínuo.

3.1.1 Soft Actor-Critic

O algoritmo *Soft Actor-Critic* (SAC) é projetado para lidar com espaços de ação contínuos, tornando-o uma escolha eficaz para tarefas de robótica que envolvem controle [Haarnoja et al., 2018b]. Ele se baseia em uma abordagem Ator-Crítica estocástica

Figura 4: Sistema da Depth-CUPRL



Fonte: Autor

e é considerado um método de política-*off*, o que significa que ele aprende a partir de políticas passadas [Haarnoja et al., 2018a]. A principal ideia do SAC é usar funções de aproximação para aprender políticas de entropia máxima. Ele alterna entre avaliar e melhorar a política, buscando encontrar a função de valor V adequada e atualizando a distribuição da política. Para isso, ele utiliza três redes neurais: uma para aproximar a política (ator), outra para estimar o valor do estado V e outra para estimar o valor de ação Q (crítica). Estas redes calculam a previsão de ação para o estado atual e geram sinais de erro de diferença temporal a cada intervalo de tempo.

Além de buscar a maximização das recompensas do sistema, o algoritmo Soft Actor-Critic (SAC) também busca maximizar a entropia da política. A entropia se refere à imprevisibilidade de uma variável. Quanto maior a entropia, maior a incerteza. Se uma variável aleatória assume apenas um único valor, então a estratégia tem entropia zero, o que incentiva a exploração do agente.

É usado de uma memória de repetição para treinar e avaliar a função de política gerada pela rede do ator, a função de valor V e a função Q são geradas pela rede crítica, de acordo com as recomendações de [Schaul et al., 2015]. A memória de repetição permite salvar as experiências do agente durante a sessão de treinamento, permitindo que sejam utilizadas para aproximar a função Q mais qualificada do problema. Essas experiências, incluindo estados, ações, recompensas e novos estados, são armazenadas em um *buffer* de memória e amostradas aleatoriamente para o aprendizado do algoritmo. Isso ajuda a interromper as correlações temporais entre episódios diferentes na fase de treinamento, mesmo com memórias curtas, e é perceptível um aumento significativo no desempenho dos algoritmos.

Deep-RL de política-*off*. Independentemente do aumento nas capacidades da aplicação com Deep-RL, o uso de memórias de repetição pode afetar o tempo de aprendizagem de um agente.

Apesar do uso de memórias de repetição, os algoritmos de *Q-learning* fazem uso de uma rede alvo [Achiam, 2018]. A expressão usa o termo alvo devido ao fato de que, ao minimizar a perda, torna a função Q mais próxima de seu alvo. No entanto, o alvo depende exatamente dos mesmos parâmetros para o treinamento θ . A variável θ representa os pesos da rede principal. Assim, a solução foi utilizar outra rede para o alvo com um retardo de tempo, chamada rede alvo. A atualização dos pesos da rede alvo pode ser definida como:

$$\theta' \leftarrow \tau\theta + (1 - \tau)\theta' \quad (9)$$

com $\tau \ll 1$. Isso nos mostra que a rede alvo é uma réplica da rede principal, utilizando a técnica de Polyak-averaging [Polyak and Juditsky, 1992] para atualizar seus parâmetros, o que contribui para a estabilidade do aprendizado nos algoritmos de Deep-RL.

3.1.2 Representações Contrastivas Não Supervisionadas para Aprendizagem por Reforço

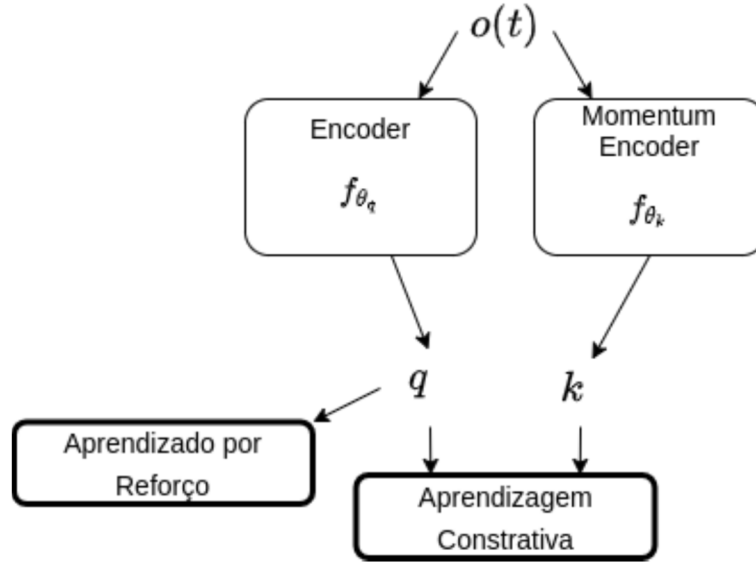
As Representações Contrastivas Não Supervisionadas para Aprendizagem por Reforço ou CURL (do inglês, *Contrastive Unsupervised Representations for Reinforcement Learning*) são uma estratégia para combinar o aprendizado contrastivo com a aprendizagem por reforço [Laskin et al., 2020]. Ela pode ser utilizada com qualquer algoritmo de RL, seja na abordagem política-*on* ou política-*off*. A abordagem SAC é frequentemente utilizada neste contexto. A CURL foi mostrada como uma técnica eficiente para melhorar a eficiência de amostragem em comparação com métodos anteriores baseados em pixel, realizando simultaneamente aprendizagem contrastiva e aprendizagem por reforço de política-*off*.

As Representações Contrastivas Não Supervisionadas para Aprendizagem por Reforço ou CURL (do inglês, *Contrastive Unsupervised Representations for Reinforcement Learning*) são uma estratégia que combina aprendizado contrastivo com aprendizagem por reforço [Laskin et al., 2020]. Essa abordagem pode ser aplicada a qualquer algoritmo de RL, tanto na abordagem política-*on* quanto na política-*off*, sendo a abordagem SAC frequentemente utilizada nesse contexto. A CURL tem se mostrado uma técnica eficiente para melhorar a eficiência de amostragem em comparação com métodos anteriores baseados em pixels, permitindo a aprendizagem contrastiva e por reforço de política-*off* de forma simultânea.

A CURL é capaz de aprender representações ricas de dados dimensionais através de aprendizagem não supervisionada contrastiva. O objetivo é garantir que uma *query* q

corresponda a uma *key* k , usando pares positivos de dados modificados a partir da mesma observação $o(t)$. A Figura 5 mostra como a CURL treina um *encoder* de representação visual para combinar as incorporações das versões modificadas dos dados o_q e o_k usando uma função de perda contrastiva do trabalho.

Figura 5: Estrutura de aprendizagem CURL



Fonte: Author

As observações da *query* o_q são tratadas pelo *encoder* f_{θ_q} , retornando $q = f_{\theta_q}(o_q)$. As *keys* são codificadas pelo *encoder* f_{θ_k} , retornando $k = f_{\theta_k}(o_k)$. As observações são construídas a partir do *batch* (i) amostrado da memória de repetição para a atualização do Deep-RL. A versão Poliak-averaging do *encoder query* é usada para atualizar θ_k , sendo $\theta_k \leftarrow \tau\theta_q + (1 - \tau)\theta_k$. A política do Deep-RL e a função de valor são construídos no topo do codificador da *query*, como mostrado na Figura 7. Esse codificador é treinado em conjunto com os objetivos do aprendizado por reforço e aprendizagem contrastiva.

$$\mathcal{L}_{oss_q} = \log \frac{\exp(q^T k_0)}{\sum_0^i \exp(q^T k_i)} \quad (10)$$

onde k_0 é a saída do *encoder* k da primeira observação recolhida do *batch* i da memória de repetição e o mesmo ocorre para k_i . Esse processo é ilustrado na atualização da rede *query* nos Algoritmos 3 e 2.

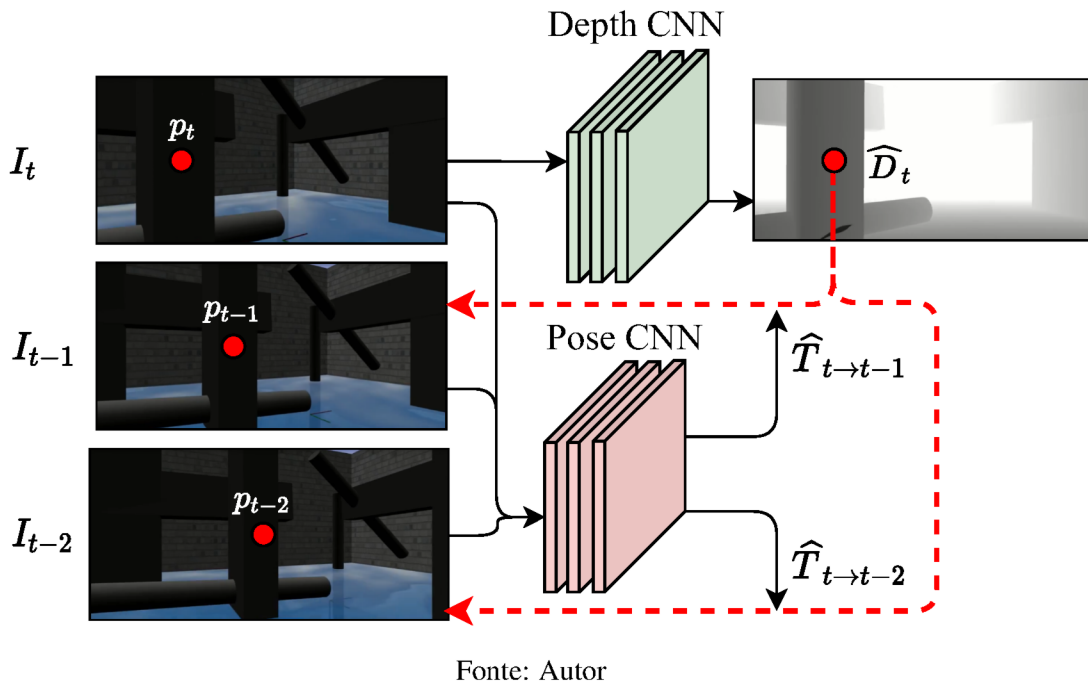
3.2 Estimativa de profundidade

Uma hipótese é que as pessoas desenvolvem uma compreensão rica e estrutural do mundo por meio de sua experiência visual anterior, que consiste em grande parte em se mover e observar um grande número de cenas e desenvolver modelos consistentes de

suas observações. A partir de milhões de tais observações, as pessoas aprendem sobre as regularidades do mundo - as estradas são planas, os edifícios são retos, os carros são sustentados por estradas etc, e podem aplicar esse conhecimento ao perceber uma nova cena, mesmo a partir de uma única imagem monocular [Yin and Shi, 2018].

Tendo em vista isso, trabalhos como de Bian et al. [2021] exploram aprender a estimar mapas de profundidade através de sequências de imagens. Sua abordagem baseia-se no *insight* de que um sistema de síntese de vista geométrica só tem um desempenho consistente e bom quando suas previsões intermediárias da geometria da cena e das poses da câmera correspondem à verdade básica física. Eles propõem uma estrutura para treinar conjuntamente uma CNN de profundidade de visão única e uma CNN de estimativa de pose de câmera a partir de sequências de vídeo não rotuladas, mostrado na Figura 6.

Figura 6: Arquitetura para profundidade e pose CNN



Além disso, é possível utilizar sensores com câmeras estéreo capazes de capturar imagens debaixo da água, como o modelo de câmera ZED [Queiroz-Neto et al., 2004]. O modelo ZED é caracterizado por imagens estéreo 2D coloridas e informações de profundidade, e pode ser utilizado para detecção de profundidade tanto no ar quanto na água, conforme descrito em [Roser et al., 2014]. Com a visão binocular, as câmeras podem determinar a distância dos objetos ao seu redor, de 0,1 a 20 m, em ambientes internos e externos.

3.3 Repetição de Experiência Priorizadas

Na repetição de experiência o agente tem a possibilidade de usar uma memória de repetição que permite eficiência dos dados pela armazenagem de experiências passadas do agente para ter a oportunidade de reprocessar os dados mais tarde. Os dados salvos na memória se referem ao estado atual s_t , ação a_t , recompensa r_t e próximo estado s_{t+1} do agente. Além do mais, as memórias de repetição também asseguram que as atualizações sejam feitas a partir de dados razoavelmente estáveis guardados na memória, o que ajuda na convergência da rede.

Enquanto a memória de repetição permite processar as transições de dados em diferentes ordens do que se foram experimentadas, há também a possibilidade de usar a repetição priorizada [Schaul et al., 2015]. Isso permite considerar transições de estados com frequências diferentes das que foram testadas experimentalmente dependendo de suas importâncias. Isso quer dizer, qual experiência é armazenada e qual é repetida. A desvantagem desse método de repetição priorizada é a introdução de tendências.

3.4 Estrutura de rede CUPRL e Depth-CUPRL

Um diagrama esquemático da arquitetura CUPRL e Depth-CUPRL é ilustrado nas Figuras 3 e 4, respectivamente. Para experimentos realizados em um contexto 2D, foi usada uma rede codificadora baseada em blocos densos, conforme descrito em [Bian et al., 2021], para gerar mapas de profundidade. Foi utilizado o dataset KITTI para treinamento dessa rede, conforme descrito em [Geiger et al., 2013]. Já para experimentos em um contexto 3D, onde podem ocorrer transições entre meios diferentes, como ar e água, foi decidido utilizar o modelo de câmera ZED para gerar os mapas de profundidade.

Para o treinamento da Depth-CUPRL, apresentado no Algoritmo 2, a estimativa de profundidade é empilhada em 3 quadros consecutivos temporais para ser usada como entrada dos encoders *query* e *key*, como mostrado na Figura 4. As imagens recebidas nos codificadores são usadas para extrair informações e retornar os espaços latentes q e k . Em seguida, as informações de posição do alvo e posição do robô no eixo z são concatenadas com o espaço latente q . Isso ocorre apenas para treinamentos e testes em ambientes com contexto 3D, já que no contexto 2D não são exploradas essas informações. A informação de posição do alvo inclui a distância polar do robô ao alvo d_t , o ângulo *yaw* e *pitch* da frente do veículo em relação ao alvo. A posição do robô no eixo z , $Z_{vehicle}$, foi usada para fornecer mais informações para transições entre meios, como ar-água e água-ar. As informações concatenadas com q são passadas para a rede de aprendizado por reforço para que possa ser tomada a ação do agente. As saídas da rede Depth-CUPRL são as velocidades lineares no eixo x e z e as velocidades angulares que serão enviadas ao robô. Já para o treinamento da CUPRL, apresentado no Algoritmo 3, há uma modificação no que é utilizado como entrada nos encoders *query* e *key*, como mostrado na Figura

3. O encoder *query* recebe imagens brutas de pixels e o encoder *key* recebe estimativas de profundidade da mesma observação. As informações concatenadas seguem o mesmo processo usado na Depth-CUPRL. Isso é visado através do aprendizado contrastivo, fazer com que os *encoders* aprendam características de profundidade e de cor ao mesmo tempo.

A principal diferença entre os algoritmos CUPRL e Depth-CUPRL ocorre na atualização da rede contrastiva. No caso da Depth-CUPRL, é selecionado um conjunto de observações de profundidade d que é uma imagem $100 \times 100 \times 3$. Essas observações são processadas por uma função que recorta a observação d em um ponto aleatório para uma imagem $84 \times 84 \times 3$, gerando d_q e d_k . Já no CUPRL, é selecionado um conjunto de informações de pixel I que é uma imagem $100 \times 100 \times 9$ e de profundidade d que é uma imagem $100 \times 100 \times 3$, ambas sem sofrer modificações antes de serem processadas.

A arquitetura de rede neural da CUPRL e Depth-CUPRL usada se assemelha à rede SAC apresentada nos trabalhos de Jesus et al. [2019] e Grando et al. [2021a]. Na Figura 7 é mostrada a arquitetura incluindo os codificadores da *query* como camadas convolucionais. As redes propostas e as de comparação utilizadas neste trabalho seguem ou se assemelham à estrutura da Figura 7. A CUPRL e a Depth-CUPRL têm como entrada a imagem bruta de pixels e o mapa de profundidade atual, respectivamente, e duas imagens e estimativas anteriores do agente no ambiente como entrada da rede ator. Essa entrada é seguida por 4 camadas convolucionais, que representam o codificador, e 3 camadas de redes neurais totalmente conectadas até a camada de saída. O número de camadas e nós foi decidido seguindo de perto a estrutura de rede feita em [Laskin et al., 2020]. A camada de saída gera a velocidade angular e linear no eixo x , e, no contexto de navegação 3D, a velocidade linear no eixo z , que é enviada ao agente. No entanto, o valor das ações está entre $[-1; 1]$ devido à função tangente hiperbólica ($\tanh$) que foi usada como função de ativação. Portanto, antes do comando ser enviado ao veículo, a velocidade angular e a velocidade linear no eixo x são restringidas entre os intervalos $-0,3$ a $0,3 \text{ rad/s}$ e 0 a $0,3 \text{ m/s}$, respectivamente. Para a navegação em contexto 3D, a velocidade linear no eixo z foi restringida entre os valores $-0,3$ a $0,3 \text{ m/s}$.

3.4.1 Função de recompensa

É necessário estabelecer um sistema de recompensa e penalidade para as redes CUPRL e Depth-CUPRL. Essas recompensas e penalidades são dadas ao agente com base em uma função de recompensa, que é criada com base em conhecimento empírico durante o processo de solução do problema. Dessa forma, a rede realiza uma etapa de *feedforward* e *backpropagation* para aprender os hiperparâmetros.

Para as tarefas em contexto 2D, o objetivo era validar se as técnicas propostas seriam capazes de navegar em um ambiente realizando o desvio de obstáculos. A função de

Algoritmo 2: Representações Priorizadas Contrastivas Não Supervisionadas de Imagens de Profundidade em Aprendizado por Reforço (Depth-CUPRL)

Entradas: Parâmetros iniciais da rede ator ϕ , parâmetros da rede crítica θ_1, θ_2 , memória de repetição M , número máximo de intervalos de tempo T

Rede para imagem: mapa de profundidade gerado D , *encoder* para rede contrastiva ψ_q e ψ_k ;

Define os parâmetros das redes alvos $\theta'_1 \leftarrow \theta_1, \theta'_2 \leftarrow \theta_2$

para $t = 1$ até T **faça**

 Observa o estado atual $d = D(I_t)$, $Alvo_{posicao}$ e $Z_{veiculo}$

 Mapa de profundidade passado para $s = \psi_q(d)$

 Concatena informações $s = concatenate(s, Alvo_{posicao}, Z_{veiculo})$

 Observa estado s and seleciona ação $a \sim \pi_\phi(\cdot|s)$ e executa a no ambiente

 Observa próximo estado s' , e recompensa r

 Armazena (s, a, r, s', d) na memória M

se Atualiza rede RL **então**

 Seleciona aleatoriamente um *batch*, $B = \{(s, a, r, s')\}$ de M

 Computa a correção para o valor da ação Q :

$$y(r, s') = r + \gamma \left(\min_{i=1,2} Q_{\theta'_i}(s', a') - \alpha \log \pi_\phi(a'|s') \right)$$

 Atualiza a rede crítica:

$$\nabla_{\theta_{i=1,2}} \frac{1}{|B|} \sum_{(s,a,r,s') \in B} (Q_{\theta_i}(s, a) - y(r, s'))^2$$

 Atualiza a politica da rede ator:

$$\nabla_\phi \frac{1}{|B|} \sum_{s \in B} \left(\min_{i=1,2} Q_{\theta_i}(s, a_\phi) - \alpha \log \pi_\phi(a_\phi|s) \right)$$

 Atualiza as redes alvo:

$$\theta'_{i=1,2} \leftarrow \tau \theta'_i + (1 - \tau) \theta_i$$

fim

se Atualiza rede contrastiva **então**

 Seleciona aleatoriamente um *batch* do mapa de profundidade d ,

$B(d) = \{(s, a, r, s', d)\}$ de M

 Modificação aleatória *aug()* para *query* e *key*: $d_q = aug(d)$ e

$d_k = aug(d)$

 Computa o espaço latente *query* e *key*: $q = \psi_q(d_q)$ e $k = \psi_k(d_k)$

 Atualiza a *query*:

$$\mathcal{L}_{oss_{\psi_q}} = \log \frac{\exp(q^T k_0)}{\sum_{i \in B(d)} \exp(q^T k_i)}$$

 Atualiza a *key*:

$$\psi_k \leftarrow \tau \psi_k + (1 - \tau) \psi_q$$

fim

fim

Algoritmo 3: Representações Priorizadas Contrastivas Não Supervisionadas de Imagens de Profundidade e Pixel em Aprendizado por Reforço (CUPRL)

Entradas: Parâmetros iniciais da rede ator ϕ , parâmetros da rede crítica θ_1, θ_2 , memória de repetição M , número máximo de intervalos de tempo T

Rede para imagem: mapa de profundidade gerado D , *encoder* para rede contrastiva ψ_q e ψ_k ;

Define os parâmetros das redes alvos $\theta'_1 \leftarrow \theta_1, \theta'_2 \leftarrow \theta_2$

para $t = 1$ até T **faça**

 Observa o estado atual $I = I_t$ e $d = D(I_t)$ *Alvo_{posicao}* e $Z_{veiculo}$

 Imagem é passada pela *query* $s = \psi_q(I)$

 Concatena informações $s = concatenate(s, Alvo_{posicao}, Z_{veiculo})$

 Observa estado s and seleciona ação $a \sim \pi_\phi(\cdot|s)$ e executa a no ambiente

 Observa próximo estado s' , e recompensa r

 Armazena (s, a, r, s', d, I) na memória M

se *Atualiza rede RL* **então**

 Seleciona aleatoriamente um *batch*, $B = \{(s, a, r, s')\}$ de M

 Computa a correção para o valor da ação Q :

$$y(r, s') = r + \gamma \left(\min_{i=1,2} Q_{\theta'_i}(s', a') - \alpha \log \pi_\phi(a'|s') \right)$$

 Atualiza a rede crítica:

$$\nabla_{\theta_{i=1,2}} \frac{1}{|B|} \sum_{(s,a,r,s') \in B} (Q_{\theta_i}(s, a) - y(r, s'))^2$$

 Atualiza a politica da rede ator:

$$\nabla_\phi \frac{1}{|B|} \sum_{s \in B} \left(\min_{i=1,2} Q_{\theta_i}(s, a_\phi) - \alpha \log \pi_\phi(a_\phi|s) \right)$$

 Atualiza as redes alvo:

$$\theta'_{i=1,2} \leftarrow \tau \theta'_i + (1 - \tau) \theta_i$$

fim

se *Atualiza rede contrastiva* **então**

 Seleciona aleatoriamente um *batch* do mapa de profundidade d e I ,

$B(d, I) = \{(s, a, r, s', d, I)\}$ de M

 Computa o espaço latente *query* e *key*: $q = \psi_q(I)$ e $k = \psi_k(d)$

 Atualiza a *query*:

$$\mathcal{Loss}_{\psi_q} = \log \frac{\exp(q^T k_0)}{\sum_{i \in B(d)} \exp(q^T k_i)}$$

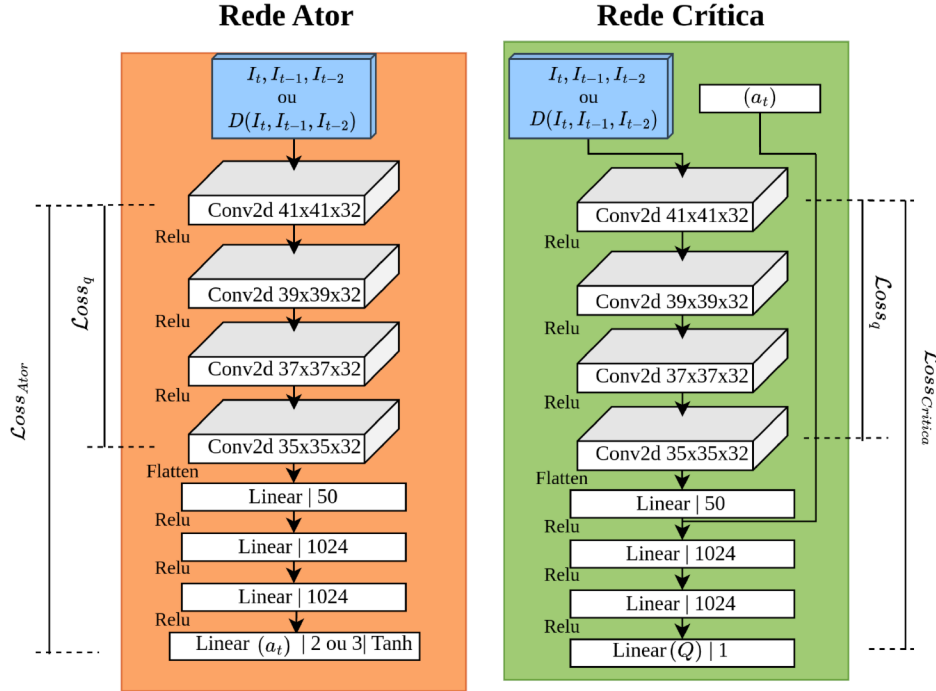
 Atualiza a *key*:

$$\psi_k \leftarrow \tau \psi_k + (1 - \tau) \psi_q$$

fim

fim

Figura 7: Arquitetura da rede CUPRL e Depth-CUPRL



Fonte: Autor

recompensa formulada para essa tarefa foi definida como:

$$r(s_t, a_t) = \begin{cases} r_{navegando} & \text{se } \min_x \geq c_o \\ r_{colidir} & \text{se } \min_x < c_o \end{cases} \quad (11)$$

onde uma recompensa negativa ($r_{colidir} = -1$) é dada se a leitura mínima de distância do robô a um objeto $\min_x$ é menor que c_o . c_o é equivalente à distância de 62cm do robô a um obstáculo ou parede que é considerado como colisão. Essas condições de colisão fazem com que o agente seja redefinido para a posição inicial de treinamento. E $r_{navegando}$ é dado quando o robô está navegando pelo mapa, cujo valor foi decidido ser 0,01. A navegação do agente é classificada pela ação no ambiente de simulação sem colidir com obstáculos. Essas recompensas foram definidas baseadas na observação de que o único objetivo é fazer com que o robô evite colidir com objetos ou paredes, sem criar uma recompensa enganosa para que o agente interprete o mapa.

Para as tarefas em contexto 3D, o objetivo é fazer a navegação até um ponto alvo de um veículo híbrido que é capaz de navegar na água e no ar. A função de recompensa

formulada para essa tarefa foi definida como:

$$r(s_t, a_t) = \begin{cases} r_{chegar} & \text{se } d_t < c_{d_t} \\ r_{colidir} & \text{se } \min_x < c_o \\ r_{navegando} = c_{navegando}(d_{t-1} - d_t) & \text{se } \min_x \geq c_o \end{cases} \quad (12)$$

onde $r_{colidir}$ e c_o seguem o mesmo propósito e valor da Equação 11. Como recompensa positiva e política que se quer otimizar através do RL, foi usado $r_{chegar} = 1$, onde essa recompensa é dada se a distância atual do robô ao alvo d_t for menor que a distância mínima do robô até o alvo $c_{d_t} = 40 \text{ cm}$, considerado como “chegar” ao ponto alvo. Porém, tratando-se de um ambiente complexo e difícil de obter a recompensa r_{chegar} , foi definida uma recompensa que tem como objetivo incentivar a aproximação entre o agente e o alvo $r_{navegando}$, onde a distância anterior e atual do robô ao alvo são usadas para gerar esse incentivo. Esse valor de incentivo é multiplicado por um valor pequeno $c_{navegando} = 0,1$, visando diminuir o impacto dessa recompensa na política principal, que é chegar ao alvo no ambiente.

Aqui, são apresentadas as propostas de arquiteturas CUPRL e Depth-CUPRL, que visam possibilitar a navegação de um veículo híbrido pelo ar e água. A Depth-CUPRL se baseia somente em observações de mapas de profundidade, enquanto a CUPRL utiliza tanto informações de profundidade quanto informações visuais de pixels.

4 RESULTADOS EXPERIMENTAIS

Esta seção apresenta e discute os resultados obtidos com as técnicas propostas neste trabalho. Os dados coletados são relacionados à recompensa alcançada pelo agente no ambiente de simulação, e esses dados permitem avaliar o desempenho do agente na tarefa de navegação. Todos os ambientes usados para o treinamento da rede foram fornecidos por Grando [2021], mas algumas modificações foram feitas no código-fonte da simulação usada do Gazebo para utilizar a câmera do veículo HUAUV simulado.

É importante observar que diferentes agentes foram treinados para cada ambiente proposto: um simples (contexto 2D) e outro mais complexo (contexto 3D). O contexto 2D e 3D referem-se a ambientes tridimensionais, no entanto, os ambientes em contexto 2D restringem o movimento do veículo a um plano, sem variação de altura, enquanto os ambientes em contexto 3D permitem o movimento livre do agente em todas as três dimensões cartesianas. Foram comparados os métodos propostos CUPRL e Depth-CUPRL com o CURL [Laskin et al., 2020] tendo como entrada a imagem bruta de pixel, e com uma versão modificada do CURL chamada de CURL (Depth) que utiliza mapas de profundidade como entrada, mas sem a utilização de memória priorizada. Também foram comparados com uma rede SAC (CNN prio.) como adotado em Grando et al. [2021b], mas com camadas convolucionais. A SAC (CNN prio.) utiliza mapas de profundidade como entradas da rede e memória priorizada. Todas as redes testadas no trabalho seguiram a arquitetura proposta na Figura 7. A posição inicial do veículo para treinamento e teste em cada um dos cenários foi estabelecida como a posição cartesiana $[3,6; -3,2; 2,5]$ para o ambiente aéreo em contexto 2D. Apenas velocidades lineares no eixo x e angulares são aplicadas ao yaw do veículo nesse caso. Para contexto 3D, a posição inicial do veículo é alterada para testes de transição ar-água e água-ar. Em testes de transição ar-água, a posição inicial do veículo é $[3,6; -3,2; 2,5]$ e para casos de transição água-ar a posição inicial do veículo é $[3,6; -3,2; -0,5]$.

4.1 Configuração do Experimento

Aqui estão descritas as ferramentas, *softwares* e ambientes empregados neste trabalho. As simulações foram rodadas com o ROS como *backbone* e com o Gazebo como simulador. Python foi escolhido como a linguagem de programação e em alguns casos C++ foi usado para executar algumas partes do algoritmo com mais rapidez. Na sua maioria os algoritmos utilizaram a biblioteca PyTorch [Rao and McMahan, 2019] para a criação das redes neurais e manipulação de imagens foi usada a biblioteca OpenCV [Bradski and Kaehler, 2008]. A plataforma robótica HUAUV utilizada em simulação foi o Hydrone, projetado por Grando [2021].

4.1.1 ROS e Gazebo

O sistema operacional de robôs (ROS) é um *middleware* de robótica de código aberto, que atua como um intermediário, gerenciando as conexões entre software e hardware em robôs. ROS [Pyo et al., 2015] não é um sistema operacional, mas fornece alguns serviços padrão de sistema operacional, como mensagens entre processos, controle de baixo nível de dispositivos, abstração de hardware e gerenciamento de pacotes. A arquitetura gráfica é utilizada para representar os processos executados pelo ROS, onde os processos são chamados “nós” e as mensagens entre dois ou mais processos se comunicando são os tópicos. As principais bibliotecas do ROS são feitas para as linguagens de programação C++, Python e LISP, assim como, para outras linguagens como Java são fornecidas como experimental [Joseph, 2015].

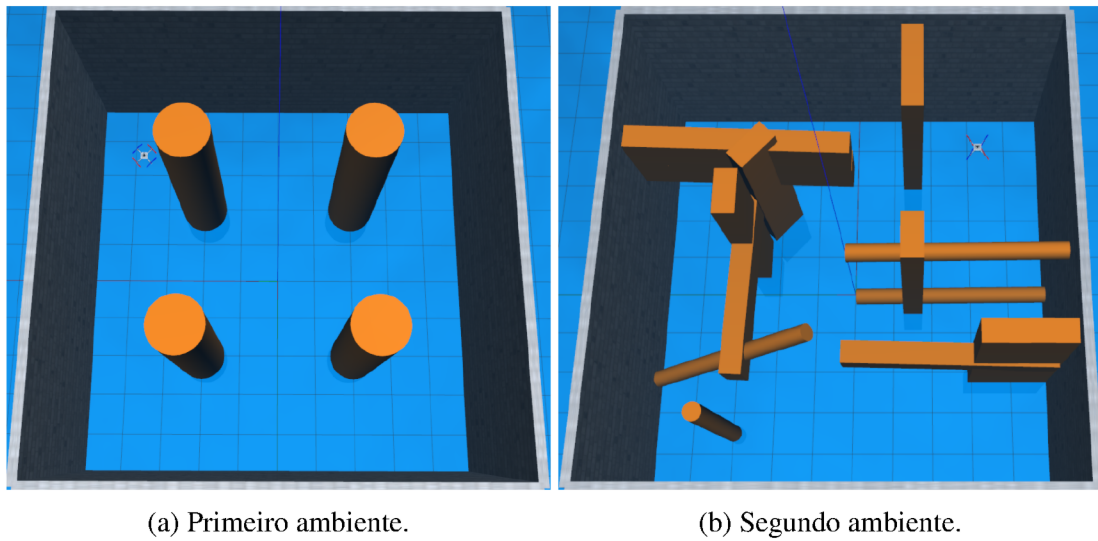
É realmente difícil criar uma aplicação para robótica sem simulá-la, portanto, a simulação é uma ferramenta essencial na caixa de ferramentas dos roboticistas. Ser capaz de projetar novos robôs e ambientes, testar algoritmos, coletar dados, aplicar modelos de inteligência artificial com facilidade e, ao mesmo tempo, com configurações realistas, todas essas características são o que faz um bom simulador. No Gazebo, todas as características acima são possíveis [Fairchild and Harman, 2016], e com a vantagem de ter uma comunidade ativa. Acompanha o ROS em sua instalação com associações totalmente implementadas.

4.1.2 Ambientes de Simulação

Os ambientes de treinamento criados no Gazebo podem ser observados na Figura 8. Eles foram projetados para demonstrar que as técnicas utilizadas neste trabalho podem navegar sem colidir com as paredes e obstáculos em um contexto 2D, e em um contexto 3D, além de transicionar entre meios aéreo e aquático, e chegar a um ponto alvo. No primeiro ambiente, há 4 obstáculos que foram alocados conforme ilustrado na Figura 8a. A adição desses obstáculos fixos produz um cenário mais desafiador para o robô. O objetivo da proposta é controlar o HUAUV com uma técnica de Deep-RL através de uma

função de recompensa que dá recompensas negativas toda vez que o robô colide com uma parede ou obstáculo. Portanto, se o veículo aéreo-aquático colidir com a parede ou qualquer obstáculo, uma recompensa negativa é dada por esta ação e o episódio atual é encerrado. Na Figura 8b, foi criado um cenário mais complexo de navegação. Esse ambiente apresenta um grau de complexidade muito grande em comparação com o primeiro. É esperado que para esse ambiente o veículo consiga navegar de forma autônoma sem colidir com obstáculos e chegar a um ponto alvo.

Figura 8: Ambiente de treinamento usado na Simulação do Gazebo.



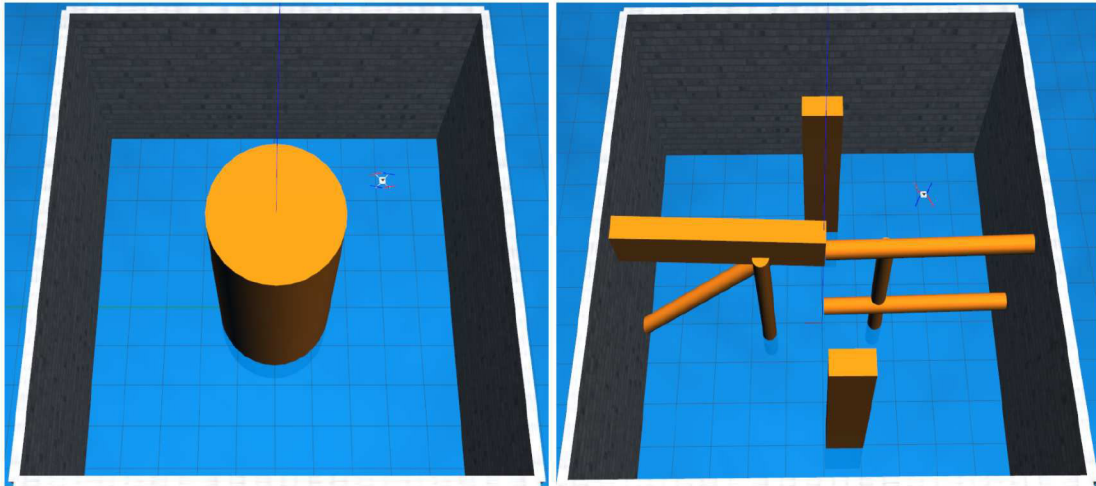
Fonte: Author

Para avaliar as eficiência das técnicas propostas neste trabalho, dois ambientes de teste foram criados, mostrados na Figura 9. O objetivo desses ambientes é avaliar o desempenho dos algoritmos após serem treinados em outros ambientes, de modo a verificar se as redes realmente aprenderam uma política de navegação eficaz. A Figura 9a apresenta um ambiente semelhante em complexidade ao primeiro ambiente de treinamento, onde as redes treinadas serão avaliadas. Já na Figura 9b, o ambiente é semelhante ao segundo ambiente de treinamento, onde as redes também serão avaliadas. Infelizmente, devido a limitações de recursos computacionais e tempo, apenas os resultados do contexto 3D foram avaliados nestes ambientes.

4.1.3 Hidrone

O veículo HUAUV usado neste trabalho foi baseado em um veículo real chamado Hydrone [Horn et al., 2019], cujo pacote de simulação foi formulado por Grando [2021]. Na Figura 10 pode-se observar o veículo real e o veículo final descrito em simulação. O veículo contém a implementação de plug-ins Gazebo e nós do ROS necessários para a simulação de UUVs e UAVs. O modelo do hydrone no ROS pode ser controlado por velocidade linear e angular em 6 graus de liberdade $(x, y, z, roll, pitch, yaw)$

Figura 9: Ambiente de avaliação usado na Simulação do Gazebo



(a) Primeiro ambiente.

(b) Segundo ambiente.

Fonte: Author

Figura 10: Veículo simulado baseado no veículo real Hydrone.



(a) veículo real.

(b) veículo simulado.

Fonte: Author

Para o *frame* base do veículo foi utilizado o valor de 1,97 kg para massa. Para a *collision* do *frame* foi estimada uma caixa com dimensões de 0,47 metros de lado e 0,09 metros de altura. Os quatro rotores foram colocados de forma simétrica com um ângulo de 90° entre cada e a uma distância exata de 0,27 metros em relação ao centro do veículo. Na Tabela 1 é possível ver os parâmetros para atuação do robô.

Tabela 1: Especificação feita no Hydrone

Itens	Especificação
Velocidade translacional máxima no eixo x e z	0,3 m/s
Velocidade rotacional máxima em yaw	0,3 rad/s
Massa	1,97 kg

Fonte: Autor

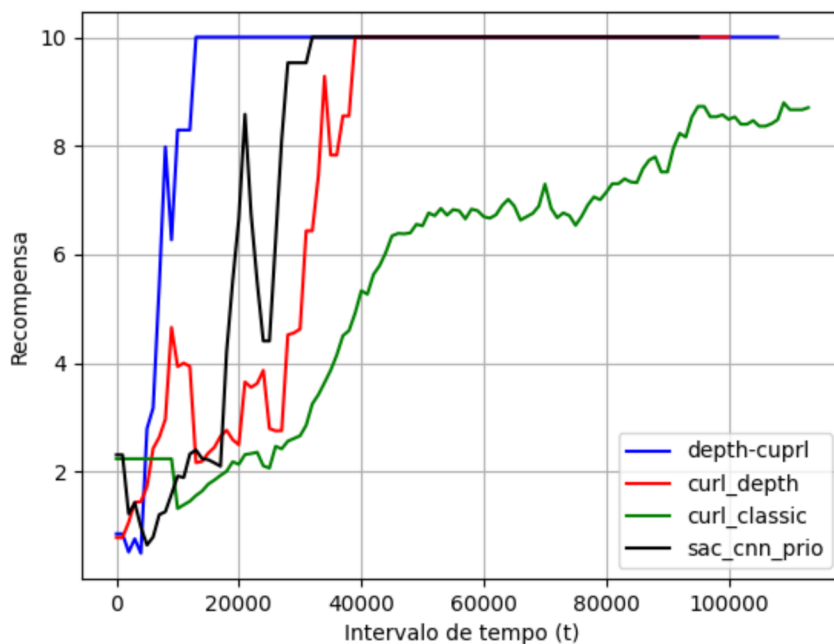
4.2 Ambiente Aéreo Contexto 2D

Para treinar no primeiro ambiente, foi configurado o ambiente com uma memória de repetição de 35.000 amostras para todas as redes treinadas. Um episódio equivale a 1000 intervalos de tempo t . Esses intervalos de tempo de ação ficam próximos a $0,025ms$ ou $40Hz$. Então, sempre que a rede neural passa por 1000 intervalos de tempo, é feita uma avaliação da rede Depth-CUPRL. A proposta CUPRL não foi avaliada para os casos de navegação em contexto 2D. A avaliação das redes é feita de modo a se utilizar apenas a resposta determinística da rede sem nenhum ruído, por meio de 10 episódios. Dessa forma, é possível avaliar a resposta da rede através de uma média de testes, tendo 10,00 como a recompensa máxima por avaliação.

Os resultados do treinamento da função de recompensa no primeiro cenário, para as redes propostas e testadas, são mostrados na Figura 11. Os métodos propostos neste trabalho foram comparados entre si, sendo o CUPRL que utiliza imagens brutas de pixel e de profundidade com memórias priorizadas, o Depth-CUPRL que usa apenas mapas de profundidade com memórias priorizadas, o CURL com entrada de imagem bruta de pixel, uma versão modificada do CURL chamada de CURL (Depth) com mapas de profundidade como entrada sem a utilização de memória priorizada e uma rede SAC (CNN prio.) com mapas de profundidade como entrada e memória priorizada. Durante os primeiros episódios, é possível observar que as redes obtêm recompensas negativas devido às ações tomadas pelo agente, pois o algoritmo ainda está aprendendo. Essas recompensas adquiridas ao longo dos episódios indicam o grau de aprendizado da técnica Deep-RL no que diz respeito à maximização da recompensa. Todas as redes foram treinadas por aproximadamente 110.000 intervalos de tempo. É possível concluir que os algoritmos Depth-CUPRL, CURL (Depth) e SAC (CNN prio) conseguiram completar a tarefa pro-

posta com sucesso. A Depth-CUPRL mostrou-se mais eficiente em termos de amostras para atingir a recompensa ótima. Já o CURL (Clássica), que utiliza imagens de pixels brutos como entrada, mostrou-se instável e não conseguiu convergir para um resultado satisfatório.

Figura 11: Recompensas do primeiro ambiente simulado - Contexto 2D



Fonte: Autor

Depois de treinar a rede neural no primeiro ambiente, foi realizada uma etapa de avaliação final da rede, com 1000 episódios. A Tabela 2 mostra os resultados obtidos para todas as técnicas avaliadas.

Tabela 2: Comparação entre proposta e outras redes no primeiro ambiente - Contexto 2D

Métodos	Imagem	Sucesso	Falha	Precisão (%)
Depth-CUPRL [Autor]	profundidade	1000	0	100%
CURL(Depth) [adaptação Laskin et al.,2020]	profundidade	1000	0	100%
CURL(Clássica) [Laskin et al., 2020]	pixel	856	144	85,6%
SAC(CNN prio.) [Haarnoja et al., 2018a]	profundidade	1000	0	100%

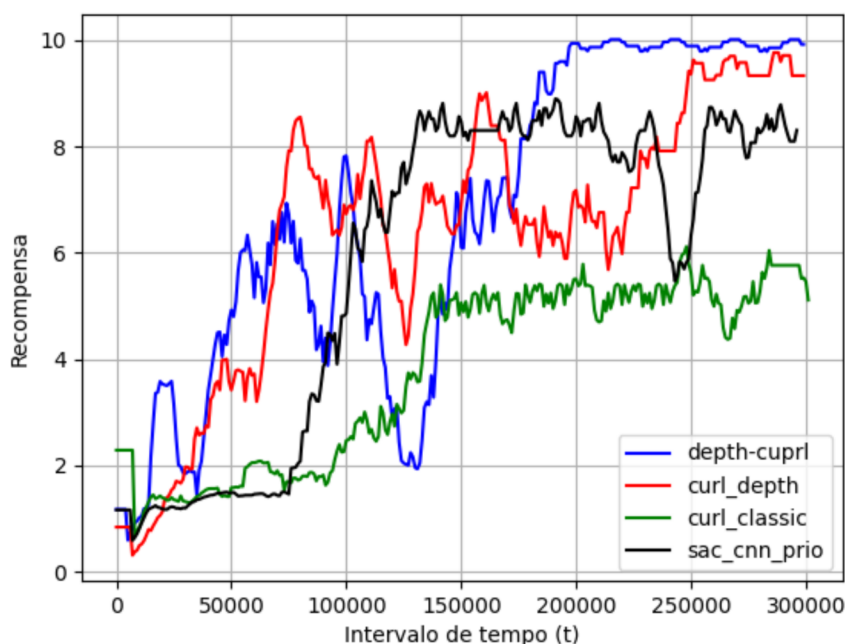
Fonte: Autor

A avaliação dos resultados obtidos no primeiro ambiente, mostrados na Tabela 2, revela que os algoritmos que utilizam informações de mapas de profundidade tiveram um desempenho significativamente melhor do que aquele que utiliza imagens baseadas em pi-

xels. Em particular, o algoritmo CURL (Clássica) apresentou muitas falhas na avaliação, enquanto a abordagem de mapas de profundidade mostrou-se mais eficiente na navegação pelo ambiente.

Os resultados da avaliação da função de recompensa do segundo ambiente de treinamento são apresentados na Figura 12. É possível observar que, devido à complexidade elevada do ambiente, foi necessário um maior número de interações de treinamento. Para o treinamento neste ambiente, foi configurado o ambiente com uma capacidade de memória de repetição de 40.000. Ao comparar com os resultados anteriores obtidos no primeiro ambiente, como mostrado na Figura 11, é possível notar que as funções de recompensa são mais instáveis para as redes neurais treinadas. Apesar disso, a técnica proposta Depth-CUPRL mostrou-se eficaz, obtendo a maior recompensa média. Isso indica a vantagem de utilizar essa técnica para a tarefa de navegação e desvio de obstáculos em ambientes complexos.

Figura 12: Recompensas do segundo ambiente simulado - Contexto 2D



Fonte: Autor

Depois de treinar as redes neurais no segundo ambiente, foi realizada uma etapa de avaliação do modelo final de treinamento das redes neurais por 1000 episódios. A Tabela 3 mostra os resultados obtidos para todas as técnicas avaliadas.

Após o treinamento das redes neurais no segundo ambiente, foi realizada uma etapa de avaliação do modelo final de treinamento das redes neurais por meio de 1000 episódios. A Tabela 3 apresenta os resultados obtidos para todas as técnicas avaliadas. Como obser-

Tabela 3: Comparação entre proposta e outras redes no segundo ambiente - Contexto 2D

Métodos	Imagem	Sucesso	Falha	Precisão (%)
Depth-CUPRL [Autor]	profundidade	992	8	99,2%
CURL (Depth) [adaptação Laskin et al.,2020]	profundidade	936	64	93,6%
CURL (Clássica) [Laskin et al., 2020]	pixel	553	447	55,3%
SAC (CNN prio.) [Haarnoja et al., 2018a]	profundidade	822	178	82,2%

Fonte: Autor

vado, as abordagens utilizando imagens de profundidade obtiveram melhores resultados, confirmando as descobertas da Tabela 2. Contudo, nenhuma das abordagens conseguiu completar o cenário sem incidentes. A abordagem Depth-CUPRL obteve o melhor desempenho, tendo travado apenas 8 vezes em 1000 tentativas, apresentando uma taxa de sucesso geral de 5,6% superior. A única abordagem que não conseguiu alcançar uma taxa de sucesso superior a 80% foi o método baseado em pixels.

Com o uso de imagens de profundidade, foi possível observar um aumento significativo no desempenho das técnicas avaliadas. O método proposto (Depth-CUPRL) obteve o melhor resultado, com uma taxa de sucesso de 43,9%, enquanto o pior método baseado em profundidade (SAC (CNN prio)) obteve uma taxa de sucesso de 26,9%. Esses resultados demonstram a eficácia de incluir informações de profundidade na abordagem de aprendizado de máquina para controlar veículos não tripulados.

4.3 Ambiente Aéreo-aquático Contexto 3D

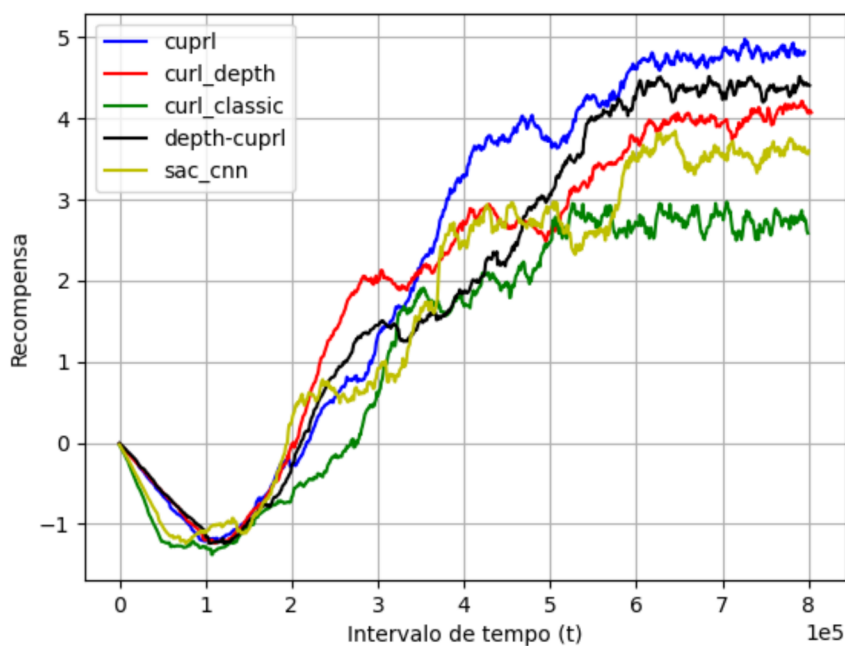
A navegação em um ambiente 3D apresenta desafios distintos devido à locomoção do veículo híbrido em três dimensões, incluindo o eixo z . Além disso, um ponto alvo é estabelecido para ser alcançado pelo agente nos ambientes de treinamento e teste. No primeiro ambiente de treinamento, o alvo é selecionado aleatoriamente e permite que o veículo navegue sem entrar em conflito com obstáculos. Esse alvo é alternado entre regiões no ar e na água de forma alternada. Desta forma, se o alvo começar no ar e o veículo conseguir chegar lá, uma recompensa é concedida à rede e o alvo é trocado para uma região na água. Se o veículo chegar ao alvo na água, o alvo é trocado para uma região no ar. Isso foi feito para forçar as transições de ambiente que o veículo precisa aprender de forma mais eficiente. O episódio de treinamento só termina em caso de colisão ou quando o número máximo de intervalos de tempo é alcançado.

Para treinar no primeiro ambiente em contexto 3D, foi configurado um ambiente com uma memória de repetição de 100.000 amostras para todas as redes treinadas. Isso foi necessário, pois ao adicionar a dimensão de locomoção no eixo z , a tarefa de alcançar um ponto alvo se tornou mais complexa. Cada episódio é composto por 1000 intervalos

de tempo t , com um intervalo de ação de $0,025ms$ ou $40Hz$. As redes neurais passam por 1000 intervalos de tempo, e então é realizada uma avaliação das redes propostas e de comparação com este trabalho. A avaliação das redes é feita utilizando apenas a resposta determinística da técnica, sem qualquer ruído, por 10 episódios.

Os resultados do treinamento da função de recompensa do primeiro ambiente, para as redes propostas e testadas em contexto 3D, são mostrados na Figura 13. É possível notar que, nos primeiros episódios de treinamento das redes neurais, há valores negativos para as recompensas das ações tomadas pelo agente. Essa característica é semelhante ao comportamento das redes treinadas no primeiro ambiente com contexto 2D. No entanto, para esse treinamento não há um valor máximo de recompensa estabelecido, pois isso depende da posição dos alvos escolhidos de forma aleatória. A recompensa adquirida nos episódios de avaliação mostra o grau de aprendizado da técnica Deep-RL e, em média, quantos alvos foram encontrados na avaliação. Isso é possível pois $r_{chegar} = 1$ é o maior valor da função de recompensa e $r_{navegando}$ é muito pequeno para impactar significativamente na média da recompensa em uma avaliação. Todas as redes foram treinadas por aproximadamente 800.000 intervalos de tempo. É possível concluir que os algoritmos propostos, CUPRL e Depth-CUPRL, obtiveram a maior média de recompensa para o primeiro ambiente de treinamento. Em seguida, as maiores recompensas foram obtidas pelas abordagens CURL (Depth), SAC (CNN prio) e CURL (Clássica).

Figura 13: Recompensas do primeiro ambiente simulado - Contexto 3D



Fonte: Autor

A avaliação da rede neural treinada no primeiro ambiente foi realizada através de 1000 episódios, divididos em duas etapas: transição ar-água e transição água-ar. Na transição ar-água, o veículo começa em uma posição inicial no ar e tem como objetivo alcançar um ponto alvo na água. Já na transição água-ar, o objetivo é alcançar um ponto alvo no ar a partir de uma posição inicial na água. As posições inicial do veículo $[x_{veiculo}; y_{veiculo}; z_{veiculo}]$ e a posição alvo $[x_{alvo}; y_{alvo}; z_{alvo}]$ são fixas e podem ser vistas na Tabela 4.

Tabela 4: Posição inicial do veículo e ponto alvo na avaliação

Posição	Ar-água	Água-ar
$[x_{veiculo}; y_{veiculo}; z_{veiculo}]$	$[3,6; -3,2; 2,5]$	$[3,6; -3,2; -0,5]$
$[x_{alvo}; y_{alvo}; z_{alvo}]$	$[-2,65; 2,65; 2,0]$	$[-2,65; 2,65; -0,5]$

Na Tabela 5 é mostrado os resultados obtidos para todas as técnicas avaliadas. É importante lembrar que as informações do *encoder key*, da Figura 3 e 4, são usadas apenas durante o treinamento da rede contrastiva e não são utilizadas durante a etapa de avaliação. Dessa forma, para o caso da CUPRL, a avaliação é realizada apenas com base em imagens brutas de pixels, como mostrado na Figura 14. O mesmo ocorre para a Depth-CUPRL, mas suas entradas incluem apenas informação de profundidade.

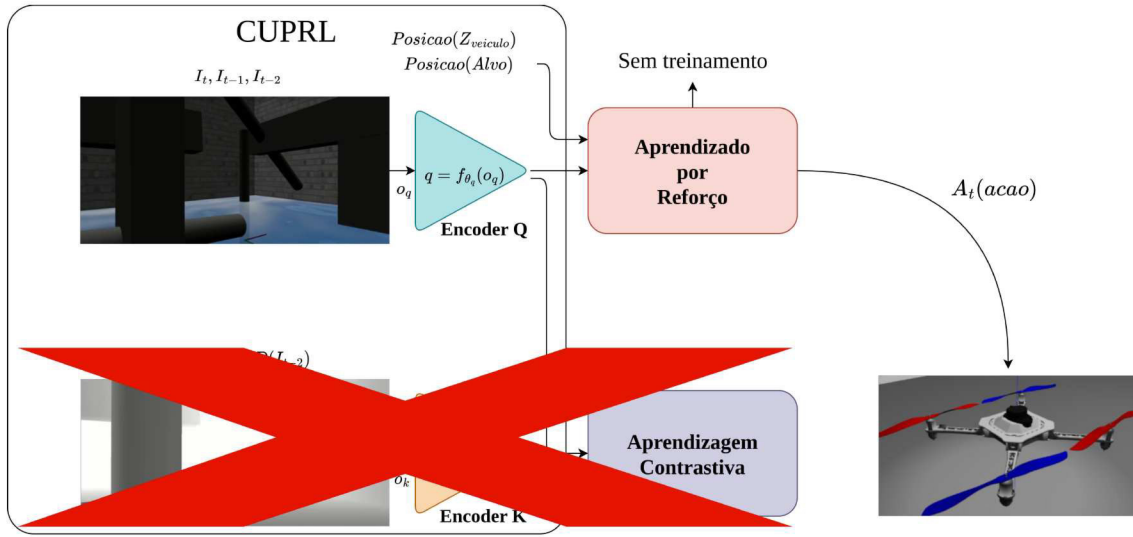
Tabela 5: Testes no primeiro ambiente de avaliação - Contexto 3D

Métodos	Imagem	Ar-água (%)	Água-ar (%)
CUPRL [Autor]	pixel	100 %	24,5 %
Depth-CUPRL [Autor]	profundidade	97,7 %	30,1 %
CURL(Depth) [adaptação Laskin et al.,2020]	profundidade	0 %	0 %
CURL(Clássica) [Laskin et al., 2020]	pixel	0 %	15,3 %
SAC(CNN prio.) [Haarnoja et al., 2018a]	profundidade	0 %	0 %

Fonte: Autor

A avaliação da Tabela 5 mostra que os algoritmos CUPRL e Depth-CUPRL obtiveram os melhores resultados. Já as outras técnicas avaliadas, não conseguiram completar a navegação até um ponto alvo ou obtiveram resultados insuficientes, como a CURL (Clássica) na transição água-ar. A transição ar-água foi onde as redes propostas se saíram melhor de acordo com testes de navegação ao alvo. É evidente que as redes mostraram uma porcentagem baixa de sucesso para chegar ao alvo na transição água-ar. Um comportamento observado nas técnicas avaliadas, especialmente aquelas que utilizam apenas mapas de profundidade, foi a dificuldade de transicionar entre meios. Para contornar este

Figura 14: Como ocorre avaliação das redes contrastivas



Fonte: Autor

problema, foi incluída uma recompensa adicional para ajudar na transição:

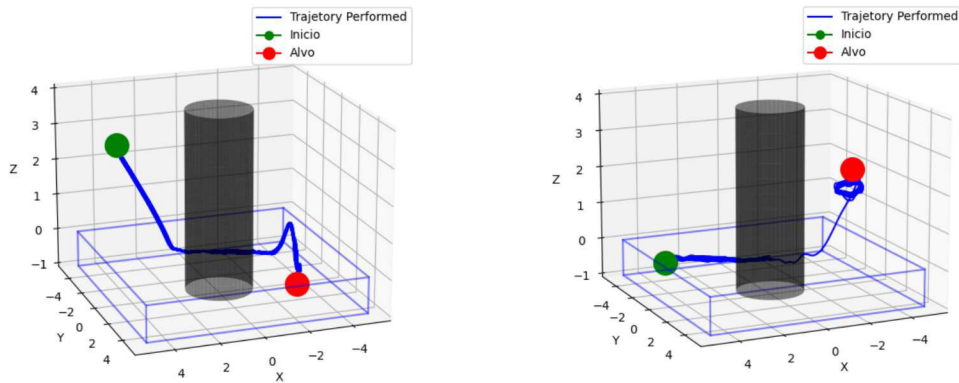
$$r_{transicao} = c_{transicao}(d_{t-1} - d_t) \text{ se } -0,05 > Z_{veiculo} > -0,23 \quad (13)$$

onde $r_{transicao}$ foi definido com o objetivo de incentivar a transição entre os meios ar-água. Esse valor de incentivo é multiplicado por um pequeno fator $c_{transicao} = 0,2$ que tem o dobro do valor de $c_{navegando}$.

Os caminhos percorridos na avaliação das redes foram registrados e são mostrados na Figura 15.

Os caminhos mostrados na Figura 15 ilustram como o veículo navegou pelo ambiente durante os 1000 episódios de avaliação. Os caminhos percorridos pela CUPRL apresen-

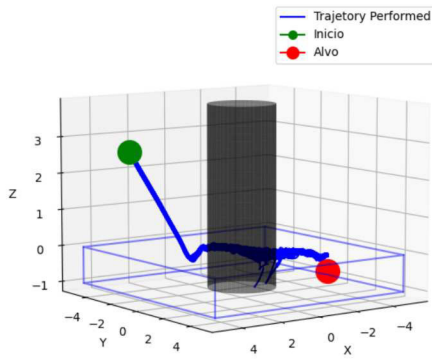
Figura 15: Caminho durante avaliação das técnicas no primeiro ambiente



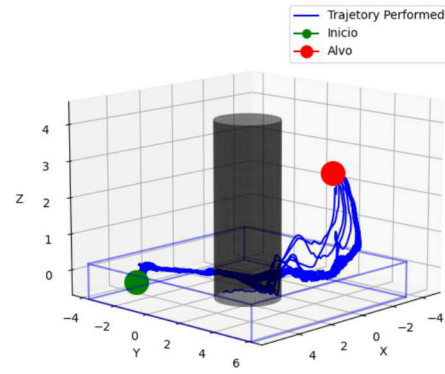
(a) CUPRL Ar-água.

(b) CUPRL Água-ar.

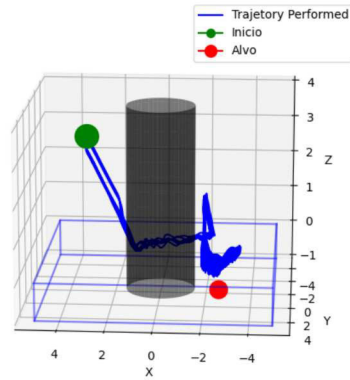
Figura continua na próxima página



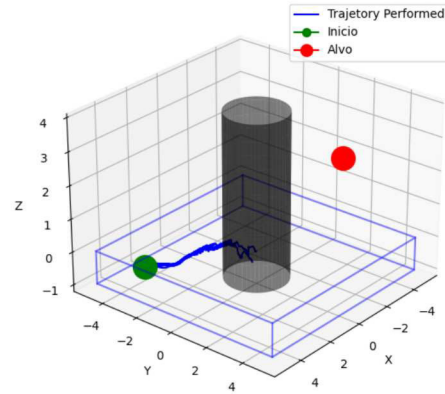
(c) Depth-CUPRL Ar-água.



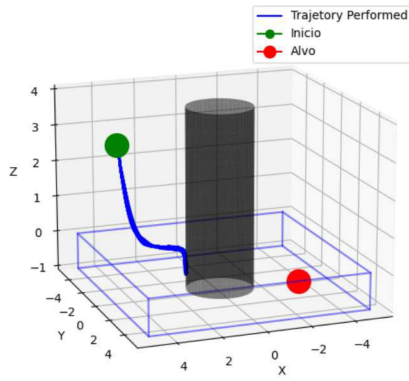
(d) Depth-CUPRL Água-ar.



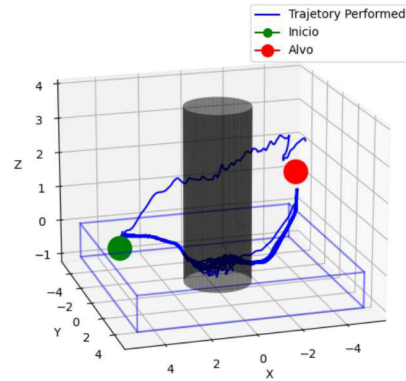
(e) CURL (Depth) Ar-água.



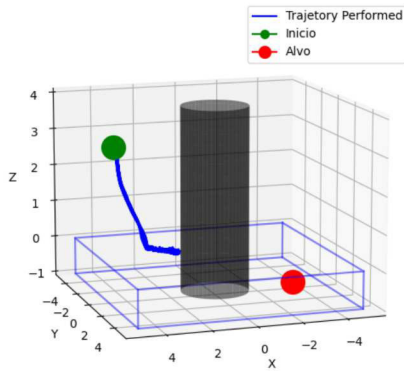
(f) CURL (Depth) Água-ar.



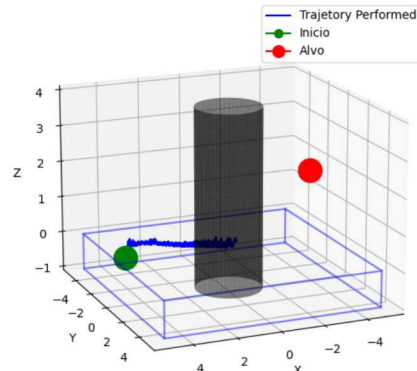
(g) CUPRL (Clássica) Ar-água.



(h) CUPRL (Clássica) Água-ar.



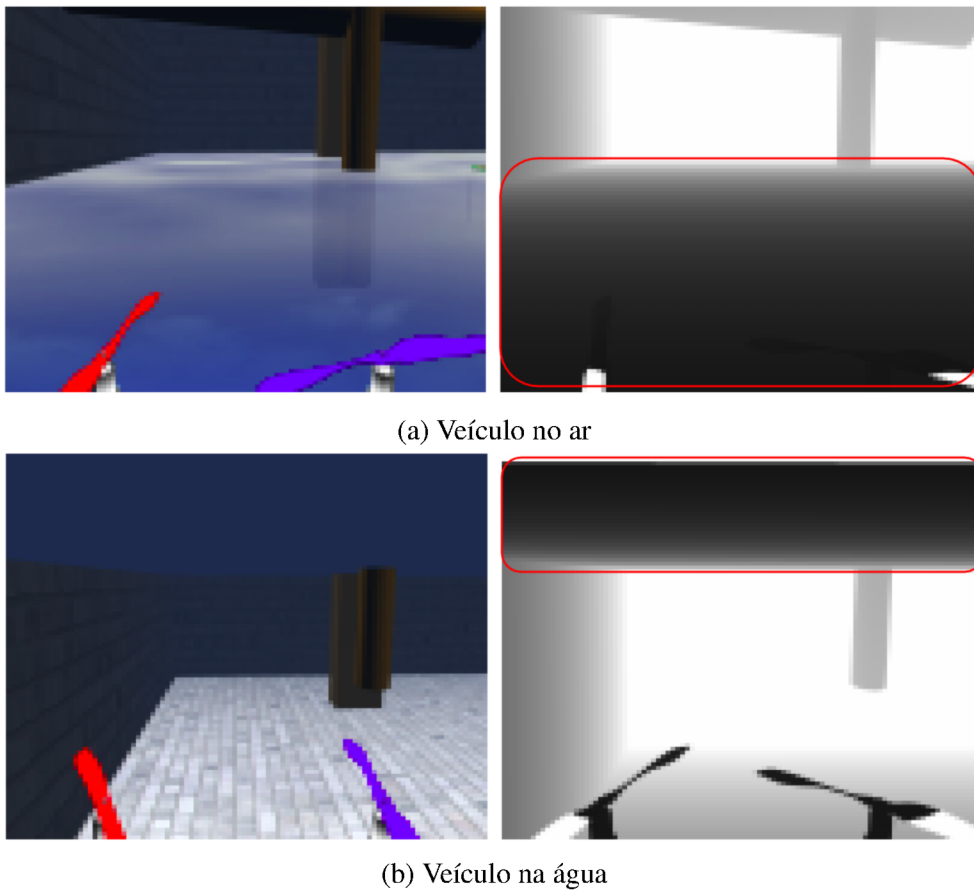
(i) SAC (CNN prio.) Ar-água.



(j) SAC (CNN prio.) Água-ar.

tam maior estabilidade na sua avaliação quando comparado com a Depth-CUPRL. Os outros métodos colidiram várias vezes, como é possível observar. No entanto, há um caso que pode ser visto na Figura 15e, onde a CURL (Depth) consegue evitar obstáculos e chegar próximo ao alvo. Entretanto, ao chegar perto, a CURL (Depth) apresenta um comportamento de *hovering* (permanecer parado no ar). Mesmo com uma recompensa para ajudar na transição, este comportamento não foi completamente eliminado. A Figura 16 pode esclarecer mais sobre essa situação. É possível ver que a água é vista como um objeto, o que pode explicar o “medo” das técnicas que usam apenas o mapa de profundidade como entrada para a navegação do veículo híbrido.

Figura 16: Imagem da água em pixel e mapa de profundidade, onde imagens da interface ar-água e água-ar são vistas como obstáculos nas imagens de profundidade

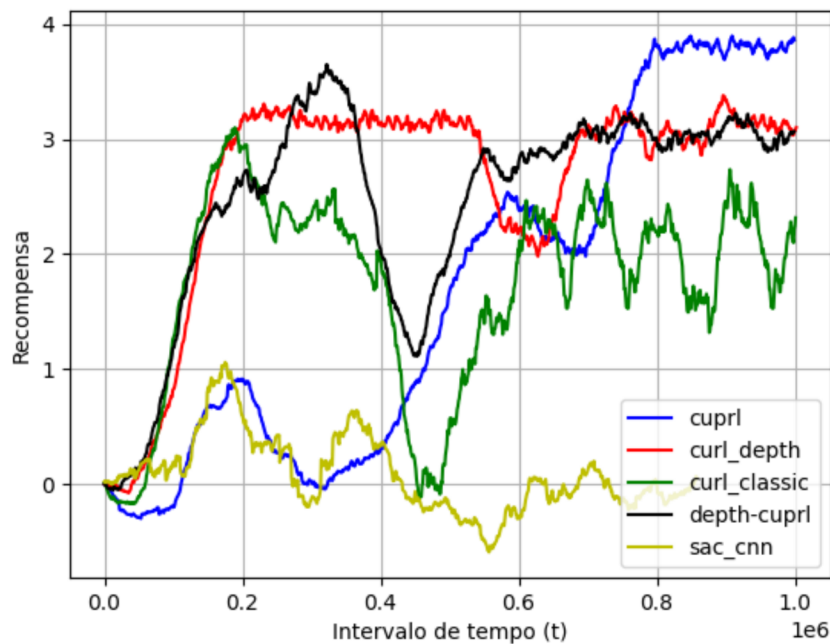


Fonte: Author

Os resultados do treinamento da função de recompensa do segundo ambiente são apresentados na Figura 13. Todas as redes foram treinadas por aproximadamente 1 milhão de intervalos de tempo. Quando comparado com as funções de recompensa anteriores, na Figura 13, é possível observar uma recompensa mais instável para os algoritmos propostos e comparados. A CUPRL, apesar de ter um treinamento mais lento no final, superou a recompensa média adquirida em comparação com os outros algoritmos. A Depth-CUPRL e CURL (Depth) apresentaram no final do treinamento resultados semelhantes entre si.

A CURL (Clássica) conseguiu obter recompensas médias boas, apesar de ter uma média instável. Já a SAC (CNN prio.), o único algoritmo que não utiliza aprendizagem contrastiva, não foi capaz de obter recompensas médias superiores a 1. Isso indica que a rede não conseguia alcançar o ponto de destino durante o treinamento da SAC (CNN prio.). Portanto, é possível concluir que a aprendizagem contrastiva apresenta uma vantagem na navegação através de imagens brutas de pixels e mapas de profundidade.

Figura 17: Recompensas do segundo ambiente simulado - Contexto 3D



Fonte: Autor

Após o treinamento das redes neurais no segundo ambiente, uma etapa de avaliação do modelo final de treinamento das redes neurais foi realizada por 1000 episódios. A posição inicial do veículo híbrido e o ponto de destino são os mesmos da Tabela 4. Os resultados da avaliação para todas as técnicas comparadas neste trabalho são apresentados na Tabela 6.

Na avaliação da Tabela 6, é possível observar que os algoritmos propostos, CUPRL e Depth-CUPRL, obtiveram novamente os melhores resultados na avaliação. Já as outras técnicas avaliadas zeraram em todos os casos. Com base nos resultados da avaliação apresentados no primeiro e segundo ambientes, é possível ver que os resultados das redes propostas se mostraram eficazes na transição ar-água. No entanto, nenhuma das redes testadas obteve bom desempenho na transição água-ar. Isso pode ser apontado como um limite dessas redes no esquemático de simulação, função de recompensa e treinamento estabelecidos.

Tabela 6: Testes no segundo ambiente de avaliação - Contexto 3D

Métodos	Imagem	Ar-Água (%)	Água-Ar (%)
CUPRL [Autor]	pixel	100 %	14,6 %
Depth-CUPRL [Autor]	profundidade	100 %	0 %
CURL (Depth) [adaptação Laskin et al.,2020]	profundidade	0 %	0 %
CURL (Clássica) [Laskin et al., 2020]	pixel	0 %	0 %
SAC (CNN prio.) [Haarnoja et al., 2018a]	profundidade	0 %	0 %

Fonte: Autor

Os caminhos percorridos durante a avaliação das redes no segundo ambiente foram registrados e apresentados na Figura 18.

Figura 18: Caminho durante avaliação das técnicas no segundo ambiente

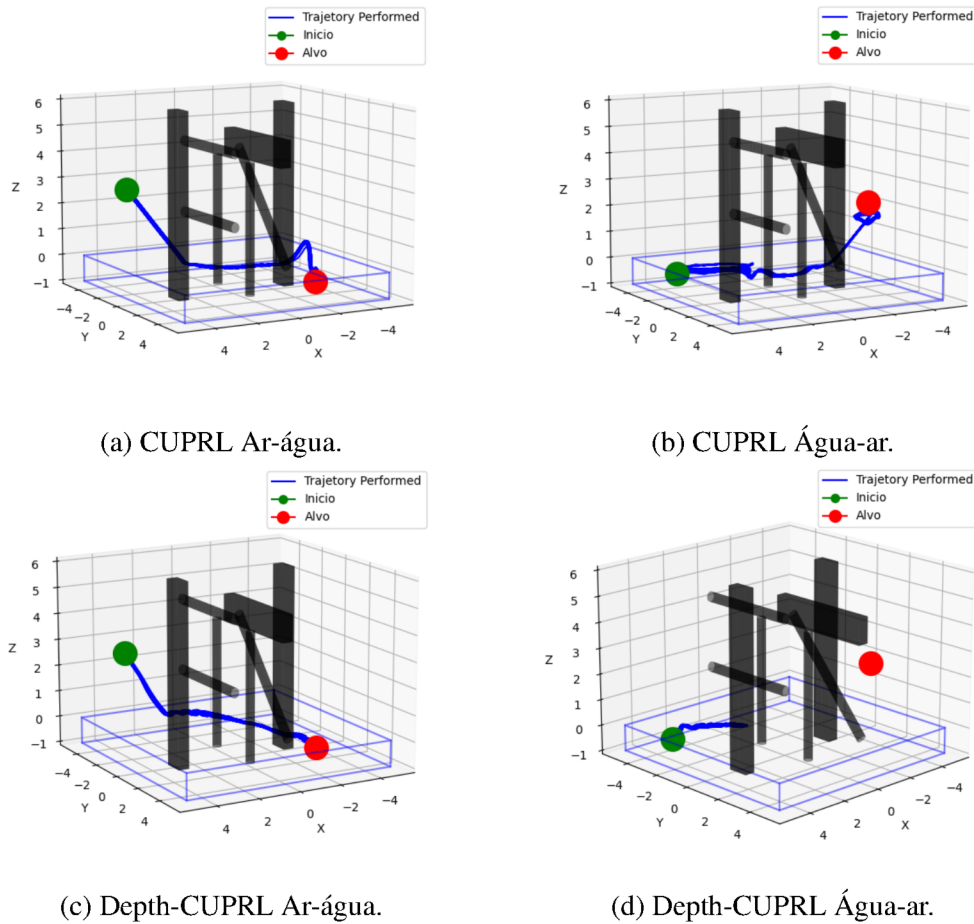
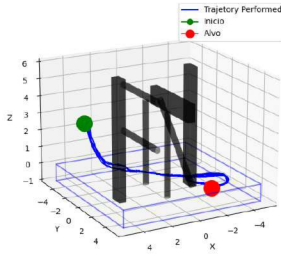
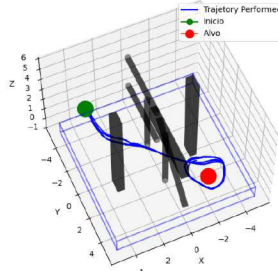


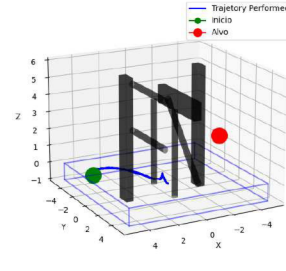
Figura continua na próxima página



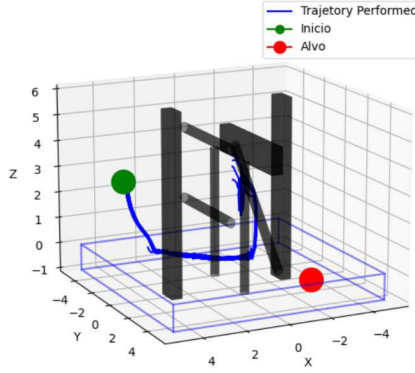
(e) CURL (Depth) Ar-água.



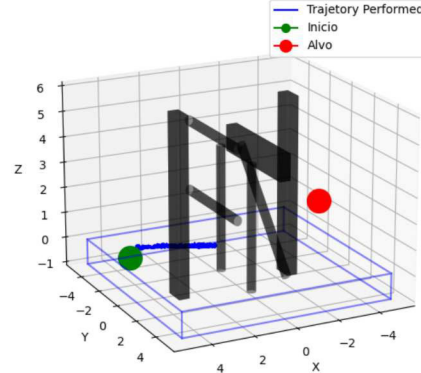
(f) CURL (Depth) Ar-água.



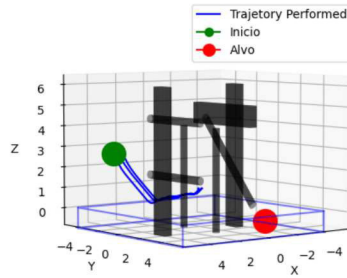
(g) CURL (Depth) Água-ar.



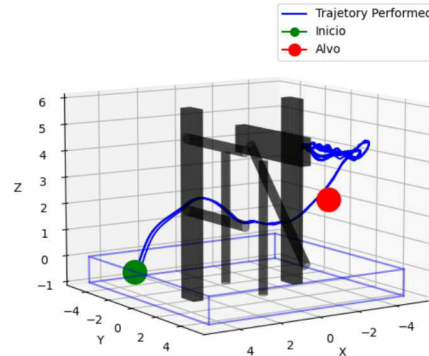
(h) CUPRL (Clássica) Ar-água.



(i) CUPRL (Clássica) Água-ar.



(j) SAC (CNN prio.) Ar-água.



(k) SAC (CNN prio.) Água-ar.

Fonte: Author

Os caminhos apresentados na Figura 18 mostram como o veículo navegou pelo segundo ambiente, que é mais complexo, durante os 1000 episódios de avaliação. A CUPRL conseguiu chegar ao destino 100% das vezes na transição ar-água e algumas vezes na transição água-ar. Já a Depth-CUPRL conseguiu navegar com sucesso somente nas transições ar-água. As outras técnicas avaliadas, CURL (Depth), CURL (Clássica) e SAC (CNN prio.), apresentaram comportamento de colisão ou "hovering" sobre a água. Este comportamento de *hovering* pode ser visto novamente na CURL (Depth). Outro comportamento interessante é o observado na avaliação da SAC (CNN prio.) na transição água-ar. Durante o treinamento, a SAC (CNN prio.) não conseguia recompensas positivas e também não apresentava uma média negativa que indicaria colisões. Esse comportamento de *hovering* próximo ao alvo pode indicar que a SAC (CNN prio.) não conseguiu

compreender a política ótima para a navegação até o ponto de destino.

5 CONSIDERAÇÕES FINAIS E PERSPECTIVAS

Este trabalho teve como objetivo desenvolver um sistema capaz de navegar um HUAUV utilizando características extraídas de imagens brutas de pixels e mapas de profundidade através de aprendizado por reforço. As abordagens propostas e outras três métodos foram desenvolvidas e aplicadas em duas tarefas em um cenário de navegação 2D e 3D. Essas etapas iniciais foram realizadas em um ambiente simulado e os resultados indicam que as redes propostas CUPRL e Depth-CUPRL apresentam melhores resultados quando comparadas com outras abordagens que visam aumentar a eficácia dos métodos de aprendizado por reforço em imagens brutas de pixels.

Como já foi demonstrado por Grando [2021], abordagens de aprendizado por reforço profundo (Deep-RL) são eficazes para a navegação de veículos HUAUV. No entanto, neste trabalho, foi investigado somente o uso de redes de Deep-RL que utilizam como entrada apenas informação de um sensor de distância e a posição do veículo. A proposta deste trabalho é investigar o uso de imagens para a navegação de veículos híbridos, tendo em vista que redes de aprendizado por reforço são ineficientes em termos de amostra com imagens [Lake et al., 2017]. A abordagem CUPRL utiliza imagens brutas de pixels e mapas de profundidade através de uma rede que aprende estas representações a partir de sequências de imagens. Em seguida, uma segunda rede é utilizada para extrair informações das imagens de pixels e estimativas de profundidade para realizar a navegação do veículo e torná-lo inteligente o suficiente para evitar colisões com objetos em um contexto 3D. A abordagem Depth-CUPRL também utiliza imagens, mas somente as estimativas de profundidade como entrada.

As Figuras 11 e 12 mostram que a recompensa aumenta com o número de treinos feitos no ambiente de contexto 2D. Isso indica que o método Depth-CUPRL é capaz de aprender a evitar obstáculos e alcançar uma recompensa quase máxima, mesmo em cenários complexos. Quanto aos resultados das Figuras 13 e 17, é possível observar que as redes propostas, CUPRL e Depth-CUPRL, obtêm a maior recompensa média entre os algoritmos comparados na navegação em contexto 3D. Além disso, foram os únicos capazes de completar a navegação em ambientes de avaliação novos. A principal contribuição deste trabalho foi desenvolver algoritmos capazes de aprender com informações visuais

de pixel e profundidade, e verificou-se que o uso de uma combinação de Deep-RL e Aprendizagem Contrastiva resulta em desempenho superior quando comparado com outras abordagens de Deep-RL baseadas apenas em informações visuais, mesmo quando os métodos propostos são aplicados apenas com informações de imagens brutas de pixel depois do treinamento, no caso da CUPRL. No entanto, há uma problemática na transição do veículo de água-ar. Para continuar o desenvolvimento deste trabalho, propõe-se investigar essa problemática da transição de meios água-ar e explorar outras técnicas que possam auxiliar em resultados mais satisfatórios. Além disso, seria importante investigar esses algoritmos para ambientes reais, levando em consideração as diferenças entre a simulação e a realidade, e superando os desafios decorrentes da transferência de conhecimento entre esses ambientes.

A implementação dos algoritmos propostos podem ser encontradas no repositório: https://github.com/dranaju/curl_navigation. Nesse repositório se encontra a implementação da Depth-CUPRL e CUPRL.

REFERÊNCIAS

Joshua Achiam. Spinning Up in Deep Reinforcement Learning. 2018.

Silas FR Alves, Joao M Rosario, Humberto Ferasoli Filho, LK Rincon, RA Yamasaki, and Alejandra Barrera. Conceptual bases of robot navigation modeling, control and applications. *Advances in Robot Navigation*, page 26, 2011.

Richard Bellman. A markovian decision process. *Journal of Mathematics and Mechanics*, pages 679–684, 1957.

Jia-Wang Bian, Huangying Zhan, Naiyan Wang, Zhichao Li, Le Zhang, Chunhua Shen, Ming-Ming Cheng, and Ian Reid. Unsupervised scale-consistent depth learning from video. *International Journal of Computer Vision*, pages 1–17, 2021.

Rogério Bonatti, Ratnesh Madaan, Vibhav Vineet, Sebastian Scherer, and Ashish Kapoor. Learning visuomotor policies for aerial navigation using cross-modal representations. *arXiv preprint Arxiv:1909.06993*, 2019.

Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. "O'Reilly Media, Inc.", 2008.

Jaeyoung Cho, Gino Lim, Taofeek Biobaku, Seonjin Kim, and Hamid Parsaei. Safety and security management with unmanned aerial vehicle (uav) in oil and gas industry. *Procedia manufacturing*, 3:1343–1349, 2015.

Junior C de Jesus, Victor A Kich, Alisson H Kolling, Ricardo B Grando, Rodrigo S Guerra, and Paulo LJ Drews. Depth-cuprl: Depth-imaged contrastive unsupervised prioritized representations in reinforcement learning for mapless navigation of unmanned aerial vehicles. In *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 10579–10586. IEEE, 2022.

Paulo LJ Drews, Armando Alves Neto, and Mario FM Campos. Hybrid unmanned aerial underwater vehicle: Modeling and simulation. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4637–4642. IEEE, 2014.

Carol Fairchild and Thomas L Harman. *ROS Robotics By Example*. Packt Publishing Ltd, 2016.

Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare, and Joelle Pineau. An introduction to deep reinforcement learning. *Foundations and Trends® in Machine Learning*, 11(3-4):219–354, 2018. ISSN 1935-8245. doi: 10.1561/22000000071. URL <http://dx.doi.org/10.1561/22000000071>.

Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision meets robotics: The kitti dataset. *The International Journal of Robotics Research*, 32(11):1231–1237, 2013.

Ricardo B. Grando. Aprendizado por reforço profundo para navegação sem mapa de um veículo híbrido aéreo-aquático. Master’s thesis, Universidade Federal do Rio Grande, 2021.

Ricardo B Grando, Junior C de Jesus, and Paulo LJ Drews-Jr. Deep reinforcement learning for mapless navigation of unmanned aerial vehicles. In *2020 Latin American Robotics Symposium (LARS), 2020 Brazilian Symposium on Robotics (SBR) and 2020 Workshop on Robotics in Education (WRE)*, pages 1–6. IEEE, 2020.

Ricardo B. Grando, Junior C. de Jesus, Victor A. Kich, Alisson H. Kolling, Nicolas P. Bortoluzzi, Pedro M. Pinheiro, Armando A. Neto, and Paulo L. J. Drews. Deep reinforcement learning for mapless navigation of a hybrid aerial underwater vehicle with medium transition. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1088–1094, 2021a. doi: 10.1109/ICRA48506.2021.9561188.

Ricardo Bedin Grando, Junior Costa de Jesus, Victor Augusto Kich, Alisson Henrique Kolling, Nicolas Pieper Bortoluzzi, Pedro Miranda Pinheiro, Armando Alves Neto, and Paulo Lilles Jorge Drews-Jr. Deep reinforcement learning for mapless navigation of a hybrid aerial underwater vehicle with medium transition. In *IEEE ICRA*, pages 1088–1094, 2021b. doi: 10.1109/ICRA48506.2021.9561188.

Ricardo Bedin Grando, Junior Costa de Jesus, Victor Augusto Kich, Alisson Henrique Kolling, and Paulo Lilles Jorge Drews-Jr. Double critic deep reinforcement learning for mapless 3d navigation of unmanned aerial vehicles. *Journal of Intelligent & Robotic Systems*, 104(2):1–14, 2022.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*, 2018a.

Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, and Sergey Levine. Soft actor-critic algorithms and applications. *CoRR*, abs/1812.05905, 2018b. URL <http://arxiv.org/abs/1812.05905>.

Lei He, Nabil Aouf, James F Whidborne, and Bifeng Song. Integrated moment-based LGMD and deep reinforcement learning for UAV obstacle avoidance. In *IEEE ICRA*, pages 7491–7497, 2020.

Alexandre C Horn, Pedro M Pinheiro, Cesar B Silva, Armando Alves Neto, and Paulo LJ Drews-Jr. A study on configuration of propellers for multirotor-like hybrid aerial-aquatic vehicles. In *2019 19th International Conference on Advanced Robotics (ICAR)*, pages 173–178. IEEE, 2019.

Junior C Jesus, Jair A Bottega, Marco ASL Cuadros, and Daniel FT Gamarra. Deep deterministic policy gradient for navigation of mobile robots in simulated environments. In *2019 19th International Conference on Advanced Robotics (ICAR)*, pages 362–367. IEEE, 2019.

Junior Costa de Jesus, Victor Augusto Kich, Alisson Henrique Kolling, Ricardo Bedin Grando, Marco Antonio de Souza Leite Cuadros, and Daniel Fernando Tello Gamarra. Soft actor-critic for navigation of mobile robots. *Journal of Intelligent & Robotic Systems*, 102(2):1–11, 2021.

Lentin Joseph. *Mastering ROS for robotics programming*. Packt Publishing Ltd, 2015.

Lukasz Kaiser, Mohammad Babaeizadeh, Piotr Milos, Blazej Osinski, Roy H Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, et al. Model-based reinforcement learning for atari. *arXiv preprint arXiv:1903.00374*, 2019.

Vijay R Konda and John N Tsitsiklis. Actor-critic algorithms. In *Advances in neural information processing systems*, pages 1008–1014, 2000.

Brenden M Lake, Tomer D Ullman, Joshua B Tenenbaum, and Samuel J Gershman. Building machines that learn and think like people. *Behavioral and brain sciences*, 40, 2017.

Michael Laskin, Aravind Srinivas, and Pieter Abbeel. Curl: Contrastive unsupervised representations for reinforcement learning. In *International Conference on Machine Learning*, pages 5639–5650. PMLR, 2020.

Bo Li, Zhigang Gan, Daqing Chen, and Dyachenko Sergey Aleksandrovich. Uav maneuvering target tracking in uncertain environments based on deep reinforcement learning and meta-learning. *Remote Sensing*, 12(22):3789, 2020.

Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

Marco M Maia, Parth Soni, and Francisco J Diez. Demonstration of an aerial and submersible vehicle capable of flight and underwater navigation with seamless air-water transition. *arXiv preprint arXiv:1507.01932*, 2015.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518 (7540):529–533, 2015.

Boris T Polyak and Anatoli B Juditsky. Acceleration of stochastic approximation by averaging. *SIAM journal on control and optimization*, 30(4):838–855, 1992.

YoonSeok Pyo, HanCheol Cho, RyuWoon Jung, and TaeHoon Lim. Ros robot programming. *Seoul, ROBOTIS Co*, 2015.

José P Queiroz-Neto, Rodrigo Carceroni, Wagner Barros, and Mario Campos. Underwater stereo. In *Proceedings. 17th Brazilian Symposium on Computer Graphics and Image Processing*, pages 170–177. IEEE, 2004.

Delip Rao and Brian McMahan. *Natural language processing with PyTorch: build intelligent language applications using deep learning*. "O'Reilly Media, Inc.", 2019.

Alejandro Rodriguez-Ramos, Carlos Sampedro, Hriday Bavle, Ignacio Gil Moreno, and Pascual Campoy. A deep reinforcement learning technique for vision-based autonomous multirotor landing on a moving platform. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1010–1017. IEEE, 2018.

Martin Roser, Matthew Dunbabin, and Andreas Geiger. Simultaneous underwater visibility assessment, enhancement and improved stereo. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3840–3847. IEEE, 2014.

Carlos Sampedro, Alejandro Rodriguez-Ramos, Hriday Bavle, Adrian Carrio, Paloma de la Puente, and Pascual Campoy. A fully-autonomous aerial robot for search and rescue applications in indoor environments using learning-based techniques. *Journal of Intelligent & Robotic Systems*, 95(2):601–627, 2019.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

Vidya Sudevan, Amit Shukla, and Hamad Karki. Current and future research focus on inspection of vertical structures in oil and gas industry. In *2018 18th International Conference on Control, Automation and Systems (ICCAS)*, pages 144–149. IEEE, 2018.

Richard S Sutton, Andrew G Barto, et al. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

Lei Tai and Ming Liu. Towards cognitive exploration through deep reinforcement learning for mobile robots. *arXiv preprint arXiv:1610.01733*, 2016.

Deepak-George Thomas, Daniil Olshanskyi, Karter Krueger, Tichakorn Wongpiromsarn, and Ali Jannesari. Interpretable uav collision avoidance using deep reinforcement learning. *arXiv preprint arXiv:2105.12254*, 2021.

Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

Xingbang Yang, Tianmiao Wang, Jianhong Liang, Guocai Yao, and Miao Liu. Survey on the novel hybrid aquatic–aerial amphibious aircraft: Aquatic unmanned aerial vehicle (aquauav). *Progress in Aerospace Sciences*, 74:131–151, 2015.

Zhichao Yin and Jianping Shi. Geonet: Unsupervised learning of dense depth, optical flow and camera pose. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1983–1992, 2018.

Zheng Zeng, Chenxin Lyu, Yuanbo Bi, Yufei Jin, Di Lu, and Lian Lian. Review of hybrid aerial underwater vehicle: Cross-domain mobility and transitions control. *Ocean Engineering*, 248:110840, 2022.

APÊNDICE A CUPRL

Disponível em: https://github.com/dranaju/curl_navigation

```
#!/usr/bin/env python
# Authors: Junior Costa de Jesus #

import rospy
import os
import json
import numpy as np
import random
import time
import sys
sys.path.append(os.path.dirname(os.path.abspath(os.path.
    dirname(__file__))))
from collections import deque
from std_msgs.msg import Float32
from environment_hydrone_image_depth import Env
import torch
import torch.nn.functional as F
import torch.optim as optim
from torch.distributions import Normal
import gc
import torch.nn as nn
import math
import copy
import cv2
from torch.utils.tensorboard import SummaryWriter

#---Directory Path---#
dirPath = os.path.dirname(os.path.realpath(__file__))
```

```

#*****

import torch
import numpy as np
import torch.nn as nn
# import gym
import os
from collections import deque
import random
from torch.utils.data import Dataset, DataLoader
from skimage.util.shape import view_as_windows

class eval_mode(object):
    def __init__(self, *models):
        self.models = models

    def __enter__(self):
        self.prev_states = []
        for model in self.models:
            self.prev_states.append(model.training)
            model.train(False)

    def __exit__(self, *args):
        for model, state in zip(self.models, self.prev_states):
            model.train(state)
        return False

def soft_update_params(net, target_net, tau):
    for param, target_param in zip(net.parameters(),
        target_net.parameters()):
        target_param.data.copy_(
            tau * param.data + (1 - tau) * target_param.data
        )

```

```

def set_seed_everywhere(seed):
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)
    np.random.seed(seed)
    random.seed(seed)

def module_hash(module):
    result = 0
    for tensor in module.state_dict().values():
        result += tensor.sum().item()
    return result

def make_dir(dir_path):
    try:
        os.mkdir(dir_path)
    except OSError:
        pass
    return dir_path

def preprocess_obs(obs, bits=5):
    """Preprocessing image, see https://arxiv.org/abs/1807.03039."""
    bins = 2**bits
    assert obs.dtype == torch.float32
    if bits < 8:
        obs = torch.floor(obs / 2**(8 - bits))
    obs = obs / bins
    obs = obs + torch.rand_like(obs) / bins
    obs = obs - 0.5
    return obs

class ReplayBuffer(Dataset):
    """Buffer to store environment transitions."""
    def __init__(

```

```

        self, obs_shape, action_shape,
        capacity, batch_size, device,
        image_size=84, transform=None,
        alpha=0.6, beta_start = 0.4, beta_frames=100000
    ):
self.capacity = capacity
self.batch_size = batch_size
self.device = device
self.image_size = image_size
self.transform = transform
# the proprioceptive obs is stored as float32, pixels
obs as uint8
# print('aquil', obs_shape)
# print('aquil', *obs_shape)
obs_dtype = np.float32 if len(obs_shape) == 1 else np
    .uint8
self.obses = dict()
self.obses['anchor'] = np.empty((capacity, obs_shape
    [0], obs_shape[1], obs_shape[2]), dtype=obs_dtype)
self.obses['pos'] = np.empty((capacity, obs_shape[0],
    obs_shape[1], obs_shape[2]), dtype=obs_dtype)
self.next_obses = np.empty((capacity, obs_shape[0],
    obs_shape[1], obs_shape[2]), dtype=obs_dtype)
self.actions = np.empty((capacity, action_shape),
    dtype=np.float32)
self.poses = np.empty((capacity, 3+6), dtype=np.
    float32)
self.next_poses = np.empty((capacity, 3+6), dtype=np.
    float32)
self.rewards = np.empty((capacity, 1), dtype=np.
    float32)
self.not_dones = np.empty((capacity, 1), dtype=np.
    float32)

self.idx = 0
self.last_save = 0
self.full = False

self.alpha = alpha

```

```

self.beta_start = beta_start
self.beta_frames = beta_frames
self.frame = 1 #for beta calculation
self.priorities = np.zeros((capacity,), dtype=np.
    float32)

def beta_by_frame(self, frame_idx):
    """
    Linearly increases beta from beta_start to 1 over
    time from 1 to beta_frames.

    3.4 ANNEALING THE BIAS (Paper: PER)
    We therefore exploit the flexibility of annealing the
    amount of importance-sampling
    correction over time, by defining a schedule on the
    exponent
    that reaches 1 only at the end of learning. In
    practice, we linearly anneal from its initial
    value 0 to 1
    """
    return min(1.0, self.beta_start + frame_idx * (1.0 -
        self.beta_start) / self.beta_frames)

def add(self, obs, obs_depth, pose, action, reward,
    next_obs, next_pose, done):

    max_prio = self.priorities.max() if (self.idx == 0
        and not self.full) else 1.0
    # print('max_prio', max_prio)

    np.copyto(self.obses['anchor'][self.idx], obs)
    np.copyto(self.obses['pos'][self.idx], obs_depth)
    np.copyto(self.poses[self.idx], pose)
    np.copyto(self.actions[self.idx], action)
    np.copyto(self.rewards[self.idx], reward)

```

```

np.copyto(self.next_obses[self.idx], next_obs)
np.copyto(self.next_poses[self.idx], next_pose)
np.copyto(self.not_dones[self.idx], not done)

self.priorities[self.idx] = max_prio
self.idx = (self.idx + 1) % self.capacity
self.full = self.full or self.idx == 0

def sample_proprio(self):

    idxs = np.random.randint(
        0, self.capacity if self.full else self.idx, size=
        self.batch_size
    )

    obses = self.obses[idxs]
    next_obses = self.next_obses[idxs]

    obses = torch.as_tensor(obses, device=self.device).
        float()
    poses = torch.as_tensor(self.poses[idxs], device=self
        .device)
    actions = torch.as_tensor(self.actions[idxs], device=
        self.device)
    rewards = torch.as_tensor(self.rewards[idxs], device=
        self.device)
    next_obses = torch.as_tensor(
        next_obses, device=self.device
    ).float()
    next_poses = torch.as_tensor(self.next_poses[idxs],
        device=self.device)
    not_dones = torch.as_tensor(self.not_dones[idxs],
        device=self.device)
    return obses, poses, actions, rewards, next_obses,
        next_poses, not_dones

def sample_cpc(self):
    if not self.full:
        N = self.idx

```

```

else:
    N = self.capacity

if N == self.capacity:
    prios = self.priorities
else:
    prios = self.priorities[:self.idx]

# calc  $P = p^a / \sum(p^a)$ 
probs = prios ** self.alpha
P = probs/probs.sum()

#gets the indices depending on the probability p
indices = np.random.choice(N, self.batch_size, p=P)

beta = self.beta_by_frame(self.frame)
self.frame+=1

#Compute importance-sampling weight
weights = (N * P[indices]) ** (-beta)
# normalize weights
weights /= weights.max()
weights = np.array(weights, dtype=np.float32)

# start = time.time()
# idxs = np.random.randint(
# 0, self.capacity if self.full else self.idx, size=
self.batch_size
# )
# print(type(self.obses))
obses = np.array([self.obses['anchor'][idxs] for idxs
    in indices])
next_obses = np.array([self.next_obses[idxs] for idxs
    in indices])
# pos = obses.copy()
pos = np.array([self.obses['pos'][idxs] for idxs in
    indices])

```

```

# print('-----')
# rgb = np.dstack((obses[0][0], obses[0][1], obses
    [0][2]))
# print(rgb.shape)

# cv2.imwrite(dirPath + '/observation_rgb.png', rgb)

# rgb = np.dstack((pos[0][0], pos[0][1], pos[0][2]))
# print(rgb.shape)

# cv2.imwrite(dirPath + '/observation_rgb_depth.png',
    rgb)

# obses = random_crop(obses, self.image_size)
# next_obses = random_crop(next_obses, self.
    image_size)
# pos = random_crop(pos, self.image_size)

obses = torch.as_tensor(obses, device=self.device).
    float()
next_obses = torch.as_tensor(
    next_obses, device=self.device
).float()
poses = torch.as_tensor(np.array([self.poses[idxs]
    for idxs in indices]), device=self.device)
next_poses = torch.as_tensor(np.array([self.
    next_poses[idxs] for idxs in indices]), device=
    self.device)

actions = torch.as_tensor(np.array([self.actions[idxs]
    for idxs in indices]), device=self.device)
rewards = torch.as_tensor(np.array([self.rewards[idxs]
    for idxs in indices]), device=self.device)
not_dones = torch.as_tensor(np.array([self.not_dones[
    idxs] for idxs in indices]), device=self.device)

pos = torch.as_tensor(pos, device=self.device).float
    ()
cpc_kwargs = dict(obs_anchor=obses, obs_pos=pos,

```

```

        time_anchor=None, time_pos=None)

    return obses, poses, actions, rewards, next_obses,
        next_poses, not_dones, cpc_kwargs, indices,
        weights

def save(self, save_dir):
    if self.idx == self.last_save:
        return
    path = os.path.join(save_dir, '%d_%d.pt' % (self.
        last_save, self.idx))
    payload = [
        self.obses[self.last_save:self.idx],
        self.next_obses[self.last_save:self.idx],
        self.actions[self.last_save:self.idx],
        self.rewards[self.last_save:self.idx],
        self.not_dones[self.last_save:self.idx]
    ]
    self.last_save = self.idx
    torch.save(payload, path)

def load(self, save_dir):
    chunks = os.listdir(save_dir)
    chucks = sorted(chunks, key=lambda x: int(x.split('_')
        )[0]))
    for chunk in chucks:
        start, end = [int(x) for x in chunk.split('.')[0].
            split('_')]
        path = os.path.join(save_dir, chunk)
        payload = torch.load(path)
        assert self.idx == start
        self.obses[start:end] = payload[0]
        self.next_obses[start:end] = payload[1]
        self.actions[start:end] = payload[2]
        self.rewards[start:end] = payload[3]
        self.not_dones[start:end] = payload[4]
        self.idx = end
        self.last_save = end
    print('Load_memory_until:' + str(self.idx))

```

```

def update_priorities(self, batch_indices,
    batch_priorities):
    for idx, prio in zip(batch_indices, batch_priorities)
        :
        self.priorities[idx] = abs(prio)

# def __getitem__(self, idx):
#     idx = np.random.randint(
#         0, self.capacity if self.full else self.idx, size=1
#     )
#     idx = idx[0]
#     obs = self.obses[idx]
#     action = self.actions[idx]
#     reward = self.rewards[idx]
#     next_obs = self.next_obses[idx]
#     not_done = self.not_dones[idx]

#     if self.transform:
#         obs = self.transform(obs)
#         next_obs = self.transform(next_obs)

#     return obs, action, reward, next_obs, not_done

def __len__(self):
    return self.capacity

# class FrameStack(gym.Wrapper):
#     def __init__(self, env, k):
#         gym.Wrapper.__init__(self, env)
#         self._k = k
#         self._frames = deque([], maxlen=k)
#         shp = env.observation_space.shape
#         self.observation_space = gym.spaces.Box(
#             low=0,
#             high=1,
#             shape=((shp[0] * k,) + shp[1:]),
#             dtype=env.observation_space.dtype
#         )

```

```

# self._max_episode_steps = env._max_episode_steps

# def reset(self):
#     obs = self.env.reset()
#     for _ in range(self._k):
#         self._frames.append(obs)
#     return self._get_obs()

# def step(self, action):
#     obs, reward, done, info = self.env.step(action)
#     self._frames.append(obs)
#     return self._get_obs(), reward, done, info

# def _get_obs(self):
#     assert len(self._frames) == self._k
#     return np.concatenate(list(self._frames), axis=0)

def random_crop(imgs, output_size):
    """
    Vectorized way to do random crop using sliding windows
    and picking out random ones

    args:
        imgs, batch images with shape (B,C,H,W)
    """
    # batch size
    n = imgs.shape[0]
    img_size = imgs.shape[-1]
    crop_max = img_size - output_size
    imgs = np.transpose(imgs, (0, 2, 3, 1))
    w1 = np.random.randint(0, crop_max, n)
    h1 = np.random.randint(0, crop_max, n)
    # creates all sliding windows combinations of size (
    output_size)
    windows = view_as_windows(
        imgs, (1, output_size, output_size, 1))[..., 0, :, :,
        0]
    # selects a random window for each batch element

```

```

        cropped_imgs = windows[np.arange(n), w1, h1]
        return cropped_imgs

def center_crop_image(image, output_size):
    h, w = image.shape[1:]
    new_h, new_w = output_size, output_size

    top = (h - new_h)//2
    left = (w - new_w)//2

    image = image[:, top:top + new_h, left:left + new_w]
    return image

def tie_weights(src, trg):
    assert type(src) == type(trg)
    trg.weight = src.weight
    trg.bias = src.bias

# for 84 x 84 inputs
OUT_DIM = {2: 39, 4: 43, 6: 31}
# for 64 x 64 inputs
OUT_DIM_64 = {2: 29, 4: 25, 6: 21}

class PixelEncoder(nn.Module):
    """Convolutional encoder of pixels observations."""
    def __init__(self, obs_shape, feature_dim, num_layers=2,
                  num_filters=32, output_logits=False):
        super(PixelEncoder, self).__init__()

        assert len(obs_shape) == 3
        self.obs_shape = obs_shape
        self.feature_dim = feature_dim
        self.num_layers = num_layers

        self.convs = nn.ModuleList(
            [nn.Conv2d(obs_shape[0], num_filters, 3, stride=2)

```

```

)
for i in range(num_layers - 1):
    self.convs.append(nn.Conv2d(num_filters,
                                num_filters, 3, stride=1))

out_dim = OUT_DIM_64[num_layers] if obs_shape[-1] ==
    64 else OUT_DIM[num_layers]
self.fc = nn.Linear(num_filters * out_dim * out_dim,
                    self.feature_dim)
self.ln = nn.LayerNorm(self.feature_dim)

self.outputs = dict()
self.output_logits = output_logits

def reparameterize(self, mu, logstd):
    std = torch.exp(logstd)
    eps = torch.randn_like(std)
    return mu + eps * std

def forward_conv(self, obs):
    obs = obs / 255.
    self.outputs['obs'] = obs

    conv = torch.relu(self.convs[0](obs))
    self.outputs['conv1'] = conv

    for i in range(1, self.num_layers):
        conv = torch.relu(self.convs[i](conv))
        self.outputs['conv%s' % (i + 1)] = conv

    # print('-----')
    # print(conv.size())

    h = conv.view(conv.size(0), -1)
    return h

def forward(self, obs, detach=False):
    h = self.forward_conv(obs)

```

```

if detach:
    h = h.detach()

    h_fc = self.fc(h)
    self.outputs['fc'] = h_fc

    h_norm = self.ln(h_fc)
    self.outputs['ln'] = h_norm

    if self.output_logits:
        out = h_norm
    else:
        out = torch.tanh(h_norm)
        self.outputs['tanh'] = out

    return out

def copy_conv_weights_from(self, source):
    """Tie convolutional layers"""
    # only tie conv layers
    for i in range(self.num_layers):
        tie_weights(src=source.convs[i], trg=self.convs[i]
                    ])

# def log(self, L, step, log_freq):
# if step % log_freq != 0:
# return

# for k, v in self.outputs.items():
# L.log_histogram('train_encoder/%s_hist' % k, v, step)
# if len(v.shape) > 2:
# L.log_image('train_encoder/%s_img' % k, v[0], step)

# for i in range(self.num_layers):
# L.log_param('train_encoder/conv%s' % (i + 1), self.
#             convs[i], step)
# L.log_param('train_encoder/fc', self.fc, step)
# L.log_param('train_encoder/ln', self.ln, step)

```

```

class IdentityEncoder(nn.Module):
    def __init__(self, obs_shape, feature_dim, num_layers,
                num_filters, *args):
        super().__init__()

        assert len(obs_shape) == 1
        self.feature_dim = obs_shape[0]

    def forward(self, obs, detach=False):
        return obs

    def copy_conv_weights_from(self, source):
        pass

    # def log(self, L, step, log_freq):
    # pass

_AVAILABLE_ENCODERS = {'pixel': PixelEncoder, 'identity':
                        IdentityEncoder}

def make_encoder(
    encoder_type, obs_shape, feature_dim, num_layers,
    num_filters, output_logits=False
):
    assert encoder_type in _AVAILABLE_ENCODERS
    return _AVAILABLE_ENCODERS[encoder_type](
        obs_shape, feature_dim, num_layers, num_filters,
        output_logits
    )

LOG_FREQ = 10000

def gaussian_logprob(noise, log_std):
    """Compute Gaussian log probability."""
    residual = (-0.5 * noise.pow(2) - log_std).sum(-1,

```

```

        keepdim=True)
    return residual - 0.5 * np.log(2 * np.pi) * noise.size
        (-1)

def squash(mu, pi, log_pi):
    """Apply squashing function.
    See appendix C from https://arxiv.org/pdf/1812.05905.pdf
    .
    """
    mu = torch.tanh(mu)
    if pi is not None:
        pi = torch.tanh(pi)
    if log_pi is not None:
        log_pi -= torch.log(F.relu(1 - pi.pow(2)) + 1e-6).sum
            (-1, keepdim=True)
    return mu, pi, log_pi

def weight_init(m):
    """Custom weight init for Conv2D and Linear layers."""
    if isinstance(m, nn.Linear):
        nn.init.orthogonal_(m.weight.data)
        m.bias.data.fill_(0.0)
    elif isinstance(m, nn.Conv2d) or isinstance(m, nn.
        ConvTranspose2d):
        # delta-orthogonal init from https://arxiv.org/pdf/1806.05393.pdf
        assert m.weight.size(2) == m.weight.size(3)
        m.weight.data.fill_(0.0)
        m.bias.data.fill_(0.0)
        mid = m.weight.size(2) // 2
        gain = nn.init.calculate_gain('relu')
        nn.init.orthogonal_(m.weight.data[:, :, mid, mid],
            gain)

class Actor(nn.Module):
    """MLP actor network."""

```

```

def __init__(
    self, obs_shape, action_shape, hidden_dim,
        encoder_type,
    encoder_feature_dim, log_std_min, log_std_max,
        num_layers, num_filters
):
    super(Actor, self).__init__()

    self.encoder = make_encoder(
        encoder_type, obs_shape, encoder_feature_dim,
            num_layers,
            num_filters, output_logits=True
    )

    self.log_std_min = log_std_min
    self.log_std_max = log_std_max

    self.trunk = nn.Sequential(
        nn.Linear(self.encoder.feature_dim + 3+6,
            hidden_dim), nn.ReLU(),
        nn.Linear(hidden_dim, hidden_dim), nn.ReLU(),
        nn.Linear(hidden_dim, 2 * action_shape)
    )

    self.outputs = dict()
    self.apply(weight_init)

def forward(
    self, obs, pose, compute_pi=True, compute_log_pi=True
        , detach_encoder=False
):
    obs = self.encoder(obs, detach=detach_encoder)

    # print('encoder_q-----')
    # # print(obs)
    # print(type(obs))
    # print(obs.size())
    # print(pose)
    # print(type(pose))

```

```

# print (pose.size())
obs_pose = torch.cat([obs, pose], dim=1)
# print (type(obs_pose))
# print ('-----')

mu, log_std = self.trunk(obs_pose).chunk(2, dim=-1)

# constrain log_std inside [log_std_min, log_std_max]
log_std = torch.tanh(log_std)
log_std = self.log_std_min + 0.5 * (
    self.log_std_max - self.log_std_min
) * (log_std + 1)

self.outputs['mu'] = mu
self.outputs['std'] = log_std.exp()

if compute_pi:
    std = log_std.exp()
    noise = torch.randn_like(mu)
    pi = mu + noise * std
else:
    pi = None
    entropy = None

if compute_log_pi:
    log_pi = gaussian_logprob(noise, log_std)
else:
    log_pi = None

mu, pi, log_pi = squash(mu, pi, log_pi)

return mu, pi, log_pi, log_std

# def log(self, L, step, log_freq=LOG_FREQ):
# if step % log_freq != 0:
# return

# for k, v in self.outputs.items():
# L.log_histogram('train_actor/%s_hist' % k, v, step)

```

```

# L.log_param('train_actor/fc1', self.trunk[0], step)
# L.log_param('train_actor/fc2', self.trunk[2], step)
# L.log_param('train_actor/fc3', self.trunk[4], step)

```

```

class QFunction(nn.Module):
    """MLP for q-function."""
    def __init__(self, obs_dim, action_dim, hidden_dim):
        super(QFunction, self).__init__()

        self.trunk = nn.Sequential(
            nn.Linear(obs_dim + 3+6 + action_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim), nn.ReLU(),
            nn.Linear(hidden_dim, 1)
        )

    def forward(self, obs, pose, action):
        assert obs.size(0) == action.size(0)

        obs_pose = torch.cat([obs, pose], dim=1)
        obs_action = torch.cat([obs_pose, action], dim=1)
        return self.trunk(obs_action)

```

```

class Critic(nn.Module):
    """Critic network, employs two q-functions."""
    def __init__(
        self, obs_shape, action_shape, hidden_dim,
        encoder_type,
        encoder_feature_dim, num_layers, num_filters
    ):
        super(Critic, self).__init__()

        self.encoder = make_encoder(
            encoder_type, obs_shape, encoder_feature_dim,
            num_layers,

```

```

        num_filters, output_logits=True
    )

    self.Q1 = QFunction(
        self.encoder.feature_dim, action_shape, hidden_dim
    )
    self.Q2 = QFunction(
        self.encoder.feature_dim, action_shape, hidden_dim
    )

    self.outputs = dict()
    self.apply(weight_init)

def forward(self, obs, pose, action, detach_encoder=
False):
    # detach_encoder allows to stop gradient propogation
    # to encoder
    obs = self.encoder(obs, detach=detach_encoder)

    # print('aquil13', obs.shape)

    q1 = self.Q1(obs, pose, action)
    q2 = self.Q2(obs, pose, action)

    self.outputs['q1'] = q1
    self.outputs['q2'] = q2

    return q1, q2

# def log(self, L, step, log_freq=LOG_FREQ):
# if step % log_freq != 0:
# return

# self.encoder.log(L, step, log_freq)

# for k, v in self.outputs.items():
# L.log_histogram('train_critic/%s_hist' % k, v, step)

# for i in range(3):

```

```
# L.log_param('train_critic/q1_fc%d' % i, self.Q1.trunk[
    i * 2], step)
# L.log_param('train_critic/q2_fc%d' % i, self.Q2.trunk[
    i * 2], step)
```

```
class CUPRL(nn.Module):
```

```
    """
```

```
    CUPRL
```

```
    """
```

```
def __init__(self, obs_shape, z_dim, batch_size, critic,
             critic_target, output_type="continuous"):
```

```
    super(CUPRL, self).__init__()
```

```
    self.batch_size = batch_size
```

```
    self.encoder = critic.encoder
```

```
    self.encoder_target = critic_target.encoder
```

```
    self.W = nn.Parameter(torch.rand(z_dim, z_dim))
```

```
    self.output_type = output_type
```

```
def encode(self, x, detach=False, ema=False):
```

```
    """
```

```
    Encoder: z_t = e(x_t)
```

```
    :param x: x_t, x y coordinates
```

```
    :return: z_t, value in r2
```

```
    """
```

```
    if ema:
```

```
        with torch.no_grad():
```

```
            z_out = self.encoder_target(x)
```

```
    else:
```

```
        z_out = self.encoder(x)
```

```
    if detach:
```

```
        z_out = z_out.detach()
```

```
    return z_out
```

```

def compute_logits(self, z_a, z_pos):
    """
    Uses logits trick for CUPRL:
    - compute (B,B) matrix z_a (W z_pos.T)
    - positives are all diagonal elements
    - negatives are all other elements
    - to compute loss use multiclass cross entropy with
      identity matrix for labels
    """
    Wz = torch.matmul(self.W, z_pos.T) # (z_dim,B)
    logits = torch.matmul(z_a, Wz) # (B,B)
    logits = logits - torch.max(logits, 1)[0][:, None]
    return logits

```

```

class CurlSacAgent(object):
    """CUPRL representation learning with SAC."""
    def __init__(
        self,
        obs_shape,
        action_shape,
        device,
        hidden_dim=256,
        discount=0.99,
        init_temperature=0.01,
        alpha_lr=1e-3,
        alpha_beta=0.9,
        actor_lr=1e-3,
        actor_beta=0.9,
        actor_log_std_min=-10,
        actor_log_std_max=2,
        actor_update_freq=2,
        critic_lr=1e-3,
        critic_beta=0.9,
        critic_tau=0.005,
        critic_target_update_freq=2,
        encoder_type='pixel',
        encoder_feature_dim=50,
        encoder_lr=1e-3,
        encoder_tau=0.005,

```

```

num_layers=4,
num_filters=32,
cpc_update_freq=1,
log_interval=100,
detach_encoder=False,
curl_latent_dim=128
):
    self.device = device
    self.discount = discount
    self.critic_tau = critic_tau
    self.encoder_tau = encoder_tau
    self.actor_update_freq = actor_update_freq
    self.critic_target_update_freq =
        critic_target_update_freq
    self.cpc_update_freq = cpc_update_freq
    self.log_interval = log_interval
    self.image_size = obs_shape[-1]
    self.curl_latent_dim = curl_latent_dim
    self.detach_encoder = detach_encoder
    self.encoder_type = encoder_type

    self.actor = Actor(
        obs_shape, action_shape, hidden_dim, encoder_type,
        encoder_feature_dim, actor_log_std_min,
        actor_log_std_max,
        num_layers, num_filters
    ).to(device)

    self.critic = Critic(
        obs_shape, action_shape, hidden_dim, encoder_type,
        encoder_feature_dim, num_layers, num_filters
    ).to(device)

    self.critic_target = Critic(
        obs_shape, action_shape, hidden_dim, encoder_type,
        encoder_feature_dim, num_layers, num_filters
    ).to(device)

    self.critic_target.load_state_dict(self.critic.

```

```

state_dict())

# tie encoders between actor and critic, and CUPRL
and critic
self.actor.encoder.copy_conv_weights_from(self.critic
.encoder)

self.log_alpha = torch.tensor(np.log(init_temperature
)).to(device)
self.log_alpha.requires_grad = True
# set target entropy to  $-|A|$ 
self.target_entropy = -np.prod(action_shape)

# optimizers
self.actor_optimizer = torch.optim.Adam(
    self.actor.parameters(), lr=actor_lr, betas=(
        actor_beta, 0.999)
)

self.critic_optimizer = torch.optim.Adam(
    self.critic.parameters(), lr=critic_lr, betas=(
        critic_beta, 0.999)
)

self.log_alpha_optimizer = torch.optim.Adam(
    [self.log_alpha], lr=alpha_lr, betas=(alpha_beta,
        0.999)
)

if self.encoder_type == 'pixel':
    # create CUPRL encoder (the 128 batch size is
    probably unnecessary)
    self.CUPRL = CUPRL(obs_shape, encoder_feature_dim,
        self.curl_latent_dim, self.critic, self.
        critic_target, output_type='continuous
        ').to(self.device)

    # optimizer for critic encoder for reconstruction
    loss

```

```

        self.encoder_optimizer = torch.optim.Adam(
            self.critic.encoder.parameters(), lr=encoder_lr
        )

        self.cpc_optimizer = torch.optim.Adam(
            self.CUPRL.parameters(), lr=encoder_lr
        )
        self.cross_entropy_loss = nn.CrossEntropyLoss()

        self.train()
        self.critic_target.train()

    def train(self, training=True):
        self.training = training
        self.actor.train(training)
        self.critic.train(training)
        if self.encoder_type == 'pixel':
            self.CUPRL.train(training)

    @property
    def alpha(self):
        return self.log_alpha.exp()

    def select_action(self, obs, pose):
        # if obs.shape[-1] != self.image_size:
        #     obs = center_crop_image(obs, self.image_size)

        with torch.no_grad():
            obs = torch.FloatTensor(obs).to(self.device)
            pose = torch.FloatTensor([pose]).to(self.device)
            obs = obs.unsqueeze(0)
            mu, _, _, _ = self.actor(obs, pose, compute_log_pi=False)
            return mu.cpu().data.numpy().flatten()

    def sample_action(self, obs, pose):
        # if obs.shape[-1] != self.image_size:
        #     obs = center_crop_image(obs, self.image_size)

```

```

with torch.no_grad():
    obs = torch.FloatTensor(obs).to(self.device)
    pose = torch.FloatTensor([pose]).to(self.device)
    obs = obs.unsqueeze(0)
    mu, pi, _, _ = self.actor(obs, pose,
                               compute_log_pi=False)
    return pi.cpu().data.numpy().flatten()

def update_critic(self, obs, pose, action, reward,
next_obs, next_pose, not_done, step, weights, writer)
:
    weights = torch.FloatTensor(weights).unsqueeze(1).to(
        self.device)

    with torch.no_grad():
        _, policy_action, log_pi, _ = self.actor(next_obs,
            next_pose)
        target_Q1, target_Q2 = self.critic_target(next_obs
            , next_pose, policy_action)
        target_V = torch.min(target_Q1,
                               target_Q2) - self.alpha.detach() *
                               log_pi
        target_Q = reward + (not_done * self.discount *
            target_V)

    # get current Q estimates
    current_Q1, current_Q2 = self.critic(
        obs, pose, action, detach_encoder=self.
            detach_encoder)
    # critic_loss = F.mse_loss(current_Q1,
    # target_Q) + F.mse_loss(current_Q2, target_Q)
    td_error1 = target_Q.detach() - current_Q1
    td_error2 = target_Q.detach() - current_Q2
    critic1_loss = 0.5 * (td_error1.pow(2) * weights).mean()
    critic2_loss = 0.5 * (td_error2.pow(2).to(self.device)
        * weights).mean()
    prios = abs((td_error1 + td_error2) / 2.0 + 1e-5).
        squeeze())

```

```

critic_loss = critic1_loss + critic2_loss

writer.add_scalar('Loss_critic', critic_loss, step)
# if step % self.log_interval == 0:
# L.log('train_critic/loss', critic_loss, step)

# Optimize the critic
self.critic_optimizer.zero_grad()
critic_loss.backward()
self.critic_optimizer.step()

# self.critic.log(L, step)
return prios

def update_actor_and_alpha(self, obs, pose, step):
    # detach encoder, so we don't update it with the
    # actor loss
    _, pi, log_pi, log_std = self.actor(obs, pose,
        detach_encoder=True)
    actor_Q1, actor_Q2 = self.critic(obs, pose, pi,
        detach_encoder=True)

    actor_Q = torch.min(actor_Q1, actor_Q2)
    actor_loss = (self.alpha.detach() * log_pi - actor_Q)
        .mean()

    # if step % self.log_interval == 0:
    # L.log('train_actor/loss', actor_loss, step)
    # L.log('train_actor/target_entropy', self.
        target_entropy, step)
    entropy = 0.5 * log_std.shape[1] * \
        (1.0 + np.log(2 * np.pi)) + log_std.sum(dim=-1)
    # if step % self.log_interval == 0:
    # L.log('train_actor/entropy', entropy.mean(),
        step)

    # optimize the actor
    writer.add_scalar('Loss_actor', actor_loss, step)

```

```

self.actor_optimizer.zero_grad()
actor_loss.backward()
self.actor_optimizer.step()

# self.actor.log(L, step)

self.log_alpha_optimizer.zero_grad()
alpha_loss = (self.alpha *
               (-log_pi - self.target_entropy).detach()).
               mean()

writer.add_scalar('Loss_alpha', alpha_loss, step)
# if step % self.log_interval == 0:
#     # L.log('train_alpha/loss', alpha_loss, step)
#     # L.log('train_alpha/value', self.alpha, step)
alpha_loss.backward()
self.log_alpha_optimizer.step()

def update_cpc(self, obs_anchor, obs_pos, cpc_kwargs,
step):

    z_a = self.CUPRL.encode(obs_anchor)
    z_pos = self.CUPRL.encode(obs_pos, ema=True)

    logits = self.CUPRL.compute_logits(z_a, z_pos)
    labels = torch.arange(logits.shape[0]).long().to(self
        .device)
    loss = self.cross_entropy_loss(logits, labels)

    self.encoder_optimizer.zero_grad()
    self.cpc_optimizer.zero_grad()
    loss.backward()

    writer.add_scalar('Loss_curl', loss, step)

    self.encoder_optimizer.step()
    self.cpc_optimizer.step()
    # if step % self.log_interval == 0:

```

```

        # L.log('train/curl_loss', loss, step)

def update(self, replay_buffer, step, writer):
    if self.encoder_type == 'pixel':
        obs, pose, action, reward, next_obs, next_pose,
        not_done, cpc_kwargs, idx, weights =
            replay_buffer.sample_cpc()
    else:
        obs, action, reward, next_obs, not_done =
            replay_buffer.sample_proprio()

    # print('-----')
    # print(cpc_kwargs["obs_anchor"].size())
    # print(action.size())

    # if step % self.log_interval == 0:
    #     L.log('train/batch_reward', reward.mean(), step)

    prios = self.update_critic(obs, pose, action, reward,
        next_obs, next_pose, not_done, step, weights,
        writer)

    replay_buffer.update_priorities(idx, prios.data.cpu()
        .numpy())

    if step % self.actor_update_freq == 0:
        self.update_actor_and_alpha(obs, pose, step)

    if step % self.critic_target_update_freq == 0:
        soft_update_params(
            self.critic.Q1, self.critic_target.Q1, self.
            critic_tau
        )
        soft_update_params(
            self.critic.Q2, self.critic_target.Q2, self.
            critic_tau
        )
        soft_update_params(

```

```

        self.critic.encoder, self.critic_target.encoder
        ,
        self.encoder_tau
    )

    if step % self.cpc_update_freq == 0 and self.
        encoder_type == 'pixel':
        obs_anchor, obs_pos = cpc_kwargs["obs_anchor"],
            cpc_kwargs["obs_pos"]
        self.update_cpc(obs_anchor, obs_pos, cpc_kwargs,
            step)

def save(self, model_dir, step):
    torch.save(
        self.actor.state_dict(), '%s/SAC_model/actor_%s.pt'
        ' % (model_dir, step)
    )
    torch.save(
        self.critic.state_dict(), '%s/SAC_model/critic_%s.
        pt' % (model_dir, step)
    )

def save_curl(self, model_dir, step):
    torch.save(
        self.CUPRL.state_dict(), '%s/SAC_model/curl_%s.pt'
        % (model_dir, step)
    )

def load(self, model_dir, step):
    self.actor.load_state_dict(
        torch.load('%s/SAC_model/actor_%s.pt' % (model_dir
        , step))
    )
    self.critic.load_state_dict(
        torch.load('%s/SAC_model/critic_%s.pt' % (
        model_dir, step))
    )
    self.CUPRL.load_state_dict(
        torch.load('%s/SAC_model/curl_%s.pt' % (model_dir,

```

```

        step))
    )

def make_agent(obs_shape, action_shape, device):
    # if args.agent == 'curl_sac':
    return CurlSacAgent(
        obs_shape=obs_shape,
        action_shape=action_shape,
        device=device,
        hidden_dim=1024,
        discount=0.99,
        init_temperature=0.1,
        alpha_lr=1e-4,
        alpha_beta=0.5,
        actor_lr=1e-3,
        actor_beta=0.9,
        actor_log_std_min=-10,
        actor_log_std_max=2,
        actor_update_freq=2,
        critic_lr=1e-3,
        critic_beta=0.9,
        critic_tau=0.01,
        critic_target_update_freq=2,
        encoder_type='pixel',
        encoder_feature_dim=50,
        encoder_lr=1e-3,
        encoder_tau=0.05,
        num_layers=4,
        num_filters=32,
        log_interval=100,
        detach_encoder=False,
        curl_latent_dim=170

    )
    # else:
    # assert 'agent is not supported: %s' % args.agent

```

```

#-----

```

```

ACTION_V_MIN = -0.3 # m/s
ACTION_W_MIN = -0.3 # rad/s
ACTION_V_MAX = 0.3 # m/s
ACTION_W_MAX = 0.3 # rad/s
replay_buffer_size = 100000
#*****
is_training = True
#-----
def action_unnormalized(action, high, low):
    action = low + (action + 1.0) * 0.5 * (high - low)
    action = np.clip(action, low, high)
    return action
#*****

writer = SummaryWriter(dirPath + '/evaluations/', flush_secs
    =1, max_queue=1)

def evaluate(num_episodes=10000, encoder_type='pixel',
    image_size=100):
    all_ep_rewards = []
    print('evau')

    def run_eval_loop(sample_stochastically=True):

        prefix = 'stochastic_' if sample_stochastically else
            ''

        if sample_stochastically:
            print('stochastically_evaluation')
        else:
            print('deterministic_evaluation')

        for i in range(num_episodes):
            obs, obs_pixel, pose = env.reset()
            pose = np.array(pose)
            # print('-print aqui-----')
            done = False
            episode_reward = 0
            step = 0

```

```

while not done:
    # if encoder_type == 'pixel':
    # obs = center_crop_image(obs, image_size)
    with eval_mode(agent):
        if sample_stochastically:
            action = agent.sample_action(obs_pixel,
                                         pose)
            unnorm_action = np.array([
                action_unnormalized(action[0],
                                    ACTION_V_MAX, 0),
                action_unnormalized(action[1],
                                    ACTION_W_MAX, ACTION_W_MIN),
                action_unnormalized(action[2],
                                    ACTION_V_MAX, ACTION_V_MIN)
            ])
        else:
            action = agent.select_action(obs_pixel,
                                         pose)
            unnorm_action = np.array([
                action_unnormalized(action[0],
                                    ACTION_V_MAX, 0),
                action_unnormalized(action[1],
                                    ACTION_W_MAX, ACTION_W_MIN),
                action_unnormalized(action[2],
                                    ACTION_V_MAX, ACTION_V_MIN)
            ])
    obs, obs_pixel, pose, reward, done = env.step(
        unnorm_action)
    pose = np.array(pose)
    # print('pose', pose)

    episode_reward += reward
    step += 1
    if step > 2:
        print('final', i, step)
        done = True
        break

all_ep_rewards.append(episode_reward)

```

```

    mean_ep_reward = np.mean(all_ep_rewards)
    best_ep_reward = np.max(all_ep_rewards)

    print('Evaluation: Mean, Best')
    print(mean_ep_reward, best_ep_reward)

    return mean_ep_reward, best_ep_reward

return run_eval_loop(sample_stochastically=False)

if __name__ == '__main__':
    rospy.init_node('curl_node')
    pub_result = rospy.Publisher('result', Float32,
        queue_size=5)
    result = Float32()
    env = Env()

    action_shape = 3
    obs_shape = (3*3, 100, 100)
    pre_aug_obs_shape = (3*3, 100, 100)

    episode, episode_reward, done = 0, 0, True
    max_steps = 1000000
    initial_step = 0
    # initial_step = 25000
    save_model_replay = False
    device = torch.device('cuda' if torch.cuda.is_available
        () else 'cpu')

    replay_buffer = ReplayBuffer(
        obs_shape=pre_aug_obs_shape,
        action_shape=action_shape,
        capacity=replay_buffer_size,
        batch_size=170,
        device=device,
        image_size=100,
    )

```

```

agent = make_agent(
    obs_shape=obs_shape,
    action_shape=action_shape,
    device=device
)

print('Path:_' + dirPath)

# agent.load(dirPath, initial_step) # 5 + 149 + 30
# replay_buffer.load(dirPath + '/replay_memory/')

for step in range(initial_step, max_steps):

    if step % 1000 == 0:
        print('start_eval')
        mean, best = evaluate()
        writer.add_scalar('Reward_mean', mean, step)
        writer.add_scalar('Reward_best', best, step)
        if save_model_replay:
            if step%(1*1000) == 0:
                agent.save(dirPath, step)
                agent.save_curl(dirPath, step)
                # replay_buffer.save(dirPath + '/'
                    replay_memory/')
                print('saved_model_and_replay_memory', step)
            # obs = env.reset()
            done = True
            save_model_replay = True

    if done:

        print("*****")
        print('Episode:_' + str(episode) + '_training')
        print('Step:_' + str(step) + '_training')
        print('Reward_average_per_ep:_' + str(
            episode_reward))
        print("*****")

```

```

obs, obs_pixel, pose = env.reset()
pose = np.array(pose)

done = False
episode_reward = 0
episode += 1

if step < 200: #1000
    print('###_Collecting_memory_###')
    action = np.array([
        np.random.uniform(-1, 1),
        np.random.uniform(-1, 1),
        np.random.uniform(-1, 1)
    ])
    unnorm_action = np.array([
        action_unnormalized(action[0], ACTION_V_MAX, 0)
        ,
        action_unnormalized(action[1], ACTION_W_MAX,
            ACTION_W_MIN),
        action_unnormalized(action[2], ACTION_V_MAX,
            ACTION_V_MIN)
    ])
else:
    with eval_mode(agent):
        action = agent.sample_action(obs_pixel, pose)
        # print('action1 ', unnorm_action)
        unnorm_action = np.array([
            action_unnormalized(action[0], ACTION_V_MAX,
                0),
            action_unnormalized(action[1], ACTION_W_MAX,
                ACTION_W_MIN),
            action_unnormalized(action[2], ACTION_V_MAX,
                ACTION_V_MIN)
        ])

if step >= (initial_step + 400): #1000
    num_updates = 1
    for _ in range(num_updates):
        # print('update')

```

```

        agent.update(replay_buffer, step, writer)

next_obs, next_obs_pixel, next_pose, reward, done =
    env.step(unnorm_action)
next_pose = np.array(next_pose)
# print('----')
# print(next_obs_pixel.max(), next_obs.max())
# print('action ', unnorm_action)
# print('pose ', next_pose)
# print('reward ', reward)
# print('----')
# print('state ', obs.shape)

episode_reward += reward
replay_buffer.add(obs_pixel, obs, pose, action,
    reward, next_obs_pixel, next_pose, done)
# print('-----')
# rgb = np.dstack((obs_pixel[0], obs_pixel[1],
#     obs_pixel[2]))
# print(rgb.shape)

# cv2.imwrite(dirPath + '/observation_rgb.png', rgb)

# rgb = np.dstack((obs[0], obs[1], obs[2]))
# print(rgb.shape)

# cv2.imwrite(dirPath + '/observation_rgb_depth.png',
#     rgb)

if reward <= -1.:
    print('\n----collide-----\n')
    # for _ in range(1):
    # # print('aqui2')
    # replay_buffer.add(obs, action, reward, next_obs,
        done)

obs = copy.deepcopy(next_obs)
obs_pixel = copy.deepcopy(next_obs_pixel)
pose = copy.deepcopy(next_pose)

```

```
# episode_step += 1
```

APÊNDICE B DEPTH-CUPRL

Disponível em: https://github.com/dranaju/curl_navigation

```
#!/usr/bin/env python
# Authors: Junior Costa de Jesus #

import rospy
import os
import json
import numpy as np
import random
import time
import sys
sys.path.append(os.path.dirname(os.path.abspath(os.path.
    dirname(__file__))))
from collections import deque
from std_msgs.msg import Float32
from environment_hydrone import Env
import torch
import torch.nn.functional as F
import torch.optim as optim
from torch.distributions import Normal
import gc
import torch.nn as nn
import math
import copy
import cv2
from torch.utils.tensorboard import SummaryWriter

#---Directory Path---#
dirPath = os.path.dirname(os.path.realpath(__file__))
```

```

#*****

import torch
import numpy as np
import torch.nn as nn
# import gym
import os
from collections import deque
import random
from torch.utils.data import Dataset, DataLoader
from skimage.util.shape import view_as_windows

class eval_mode(object):
    def __init__(self, *models):
        self.models = models

    def __enter__(self):
        self.prev_states = []
        for model in self.models:
            self.prev_states.append(model.training)
            model.train(False)

    def __exit__(self, *args):
        for model, state in zip(self.models, self.prev_states):
            model.train(state)
        return False

def soft_update_params(net, target_net, tau):
    for param, target_param in zip(net.parameters(),
        target_net.parameters()):
        target_param.data.copy_(
            tau * param.data + (1 - tau) * target_param.data
        )

```

```

def set_seed_everywhere(seed):
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)
    np.random.seed(seed)
    random.seed(seed)

def module_hash(module):
    result = 0
    for tensor in module.state_dict().values():
        result += tensor.sum().item()
    return result

def make_dir(dir_path):
    try:
        os.mkdir(dir_path)
    except OSError:
        pass
    return dir_path

def preprocess_obs(obs, bits=5):
    """Preprocessing image, see https://arxiv.org/abs/1807.03039."""
    bins = 2**bits
    assert obs.dtype == torch.float32
    if bits < 8:
        obs = torch.floor(obs / 2**(8 - bits))
    obs = obs / bins
    obs = obs + torch.rand_like(obs) / bins
    obs = obs - 0.5
    return obs

class ReplayBuffer(Dataset):
    """Buffer to store environment transitions."""
    def __init__(

```

```

        self, obs_shape, action_shape,
        capacity, batch_size, device,
        image_size=84, transform=None,
        alpha=0.6, beta_start = 0.4, beta_frames=100000
    ):
self.capacity = capacity
self.batch_size = batch_size
self.device = device
self.image_size = image_size
self.transform = transform
# the proprioceptive obs is stored as float32, pixels
obs as uint8
# print('aquil', obs_shape)
# print('aquil', *obs_shape)
obs_dtype = np.float32 if len(obs_shape) == 1 else np
    .uint8
self.obses = np.empty((capacity, obs_shape[0],
    obs_shape[1], obs_shape[2]), dtype=obs_dtype)
self.next_obses = np.empty((capacity, obs_shape[0],
    obs_shape[1], obs_shape[2]), dtype=obs_dtype)
self.actions = np.empty((capacity, action_shape),
    dtype=np.float32)
self.poses = np.empty((capacity, 3+6), dtype=np.
    float32)
self.next_poses = np.empty((capacity, 3+6), dtype=np.
    float32)
self.rewards = np.empty((capacity, 1), dtype=np.
    float32)
self.not_dones = np.empty((capacity, 1), dtype=np.
    float32)

self.idx = 0
self.last_save = 0
self.full = False

self.alpha = alpha
self.beta_start = beta_start
self.beta_frames = beta_frames
self.frame = 1 #for beta calculation

```

```

self.priorities = np.zeros((capacity,), dtype=np.
    float32)

def beta_by_frame(self, frame_idx):
    """
    Linearly increases beta from beta_start to 1 over
    time from 1 to beta_frames.

    3.4 ANNEALING THE BIAS (Paper: PER)
    We therefore exploit the flexibility of annealing the
    amount of importance-sampling
    correction over time, by defining a schedule on the
    exponent
    that reaches 1 only at the end of learning. In
    practice, we linearly anneal from its initial
    value 0 to 1
    """
    return min(1.0, self.beta_start + frame_idx * (1.0 -
        self.beta_start) / self.beta_frames)

def add(self, obs, pose, action, reward, next_obs,
    next_pose, done):

    max_prio = self.priorities.max() if (self.idx == 0
        and not self.full) else 1.0
    # print('max_prio', max_prio)

    np.copyto(self.obses[self.idx], obs)
    np.copyto(self.poses[self.idx], pose)
    np.copyto(self.actions[self.idx], action)
    np.copyto(self.rewards[self.idx], reward)
    np.copyto(self.next_obses[self.idx], next_obs)
    np.copyto(self.next_poses[self.idx], next_pose)
    np.copyto(self.not_dones[self.idx], not done)

```

```

self.priorities[self.idx] = max_prio
self.idx = (self.idx + 1) % self.capacity
self.full = self.full or self.idx == 0

def sample_proprio(self):

    idxs = np.random.randint(
        0, self.capacity if self.full else self.idx, size=
        self.batch_size
    )

    obses = self.obses[idxs]
    next_obses = self.next_obses[idxs]

    obses = torch.as_tensor(obses, device=self.device).
        float()
    poses = torch.as_tensor(self.poses[idxs], device=self
        .device)
    actions = torch.as_tensor(self.actions[idxs], device=
        self.device)
    rewards = torch.as_tensor(self.rewards[idxs], device=
        self.device)
    next_obses = torch.as_tensor(
        next_obses, device=self.device
    ).float()
    next_poses = torch.as_tensor(self.next_poses[idxs],
        device=self.device)
    not_dones = torch.as_tensor(self.not_dones[idxs],
        device=self.device)
    return obses, poses, actions, rewards, next_obses,
        next_poses, not_dones

def sample_cpc(self):
    if not self.full:
        N = self.idx
    else:
        N = self.capacity

    if N == self.capacity:

```

```

        prios = self.priorities
else:
        prios = self.priorities[:self.idx]

        # calc  $P = p^a / \sum(p^a)$ 
        probs = prios ** self.alpha
        P = probs/probs.sum()

        #gets the indices depending on the probability p
        indices = np.random.choice(N, self.batch_size, p=P)

        beta = self.beta_by_frame(self.frame)
        self.frame+=1

        #Compute importance-sampling weight
        weights = (N * P[indices]) ** (-beta)
        # normalize weights
        weights /= weights.max()
        weights = np.array(weights, dtype=np.float32)

        # start = time.time()
        # idxs = np.random.randint(
        # 0, self.capacity if self.full else self.idx, size=
        # self.batch_size
        # )
        # print(type(self.obses))
        obses = np.array([self.obses[idxs] for idxs in
            indices])
        next_obses = np.array([self.next_obses[idxs] for idxs
            in indices])
        pos = obses.copy()

        obses = random_crop(obses, self.image_size)
        next_obses = random_crop(next_obses, self.image_size)
        pos = random_crop(pos, self.image_size)

        obses = torch.as_tensor(obses, device=self.device).
float()

```

```

next_obses = torch.as_tensor(
    next_obses, device=self.device
).float()
poses = torch.as_tensor(np.array([self.poses[idxs]
    for idxs in indices]), device=self.device)
next_poses = torch.as_tensor(np.array([self.
    next_poses[idxs] for idxs in indices]), device=
    self.device)

actions = torch.as_tensor(np.array([self.actions[idxs]
    ] for idxs in indices]), device=self.device)
rewards = torch.as_tensor(np.array([self.rewards[idxs]
    ] for idxs in indices]), device=self.device)
not_dones = torch.as_tensor(np.array([self.not_dones[
    idxs] for idxs in indices]), device=self.device)

pos = torch.as_tensor(pos, device=self.device).float
    ()
cpc_kwargs = dict(obs_anchor=obses, obs_pos=pos,
    time_anchor=None, time_pos=None)

return obses, poses, actions, rewards, next_obses,
    next_poses, not_dones, cpc_kwargs, indices,
    weights

def save(self, save_dir):
    if self.idx == self.last_save:
        return
    path = os.path.join(save_dir, '%d_%d.pt' % (self.
        last_save, self.idx))
    payload = [
        self.obses[self.last_save:self.idx],
        self.next_obses[self.last_save:self.idx],
        self.actions[self.last_save:self.idx],
        self.rewards[self.last_save:self.idx],
        self.not_dones[self.last_save:self.idx]
    ]
    self.last_save = self.idx
    torch.save(payload, path)

```

```

def load(self, save_dir):
    chunks = os.listdir(save_dir)
    chucks = sorted(chunks, key=lambda x: int(x.split('_')
        )[0]))
    for chunk in chucks:
        start, end = [int(x) for x in chunk.split('.')[0].
            split('_')]
        path = os.path.join(save_dir, chunk)
        payload = torch.load(path)
        assert self.idx == start
        self.obses[start:end] = payload[0]
        self.next_obses[start:end] = payload[1]
        self.actions[start:end] = payload[2]
        self.rewards[start:end] = payload[3]
        self.not_dones[start:end] = payload[4]
        self.idx = end
        self.last_save = end
    print('Load_memory_until:' + str(self.idx))

def update_priorities(self, batch_indices,
    batch_priorities):
    for idx, prio in zip(batch_indices, batch_priorities)
        :
        self.priorities[idx] = abs(prio)

# def __getitem__(self, idx):
# idx = np.random.randint(
# 0, self.capacity if self.full else self.idx, size=1
# )
# idx = idx[0]
# obs = self.obses[idx]
# action = self.actions[idx]
# reward = self.rewards[idx]
# next_obs = self.next_obses[idx]
# not_done = self.not_dones[idx]

# if self.transform:
# obs = self.transform(obs)

```

```

    # next_obs = self.transform(next_obs)

    # return obs, action, reward, next_obs, not_done

    def __len__(self):
        return self.capacity

# class FrameStack(gym.Wrapper):
# def __init__(self, env, k):
# gym.Wrapper.__init__(self, env)
# self._k = k
# self._frames = deque([], maxlen=k)
# shp = env.observation_space.shape
# self.observation_space = gym.spaces.Box(
# low=0,
# high=1,
# shape=((shp[0] * k,) + shp[1:]),
# dtype=env.observation_space.dtype
# )
# self._max_episode_steps = env._max_episode_steps

# def reset(self):
# obs = self.env.reset()
# for _ in range(self._k):
# self._frames.append(obs)
# return self._get_obs()

# def step(self, action):
# obs, reward, done, info = self.env.step(action)
# self._frames.append(obs)
# return self._get_obs(), reward, done, info

# def _get_obs(self):
# assert len(self._frames) == self._k
# return np.concatenate(list(self._frames), axis=0)

def random_crop(imgs, output_size):
    """

```

*Vectorized way to do random crop using sliding windows
and picking out random ones*

```
args:
    imgs, batch images with shape (B,C,H,W)
    """
    # batch size
    n = imgs.shape[0]
    img_size = imgs.shape[-1]
    crop_max = img_size - output_size
    imgs = np.transpose(imgs, (0, 2, 3, 1))
    w1 = np.random.randint(0, crop_max, n)
    h1 = np.random.randint(0, crop_max, n)
    # creates all sliding windows combinations of size (
        output_size)
    windows = view_as_windows(
        imgs, (1, output_size, output_size, 1))[..., 0, :, :,
        0]
    # selects a random window for each batch element
    cropped_imgs = windows[np.arange(n), w1, h1]
    return cropped_imgs
```

```
def center_crop_image(image, output_size):
    h, w = image.shape[1:]
    new_h, new_w = output_size, output_size

    top = (h - new_h)//2
    left = (w - new_w)//2

    image = image[:, top:top + new_h, left:left + new_w]
    return image
```

```
def tie_weights(src, trg):
    assert type(src) == type(trg)
    trg.weight = src.weight
    trg.bias = src.bias
```

```
# for 84 x 84 inputs
```

```

OUT_DIM = {2: 39, 4: 35, 6: 31}
# for 64 x 64 inputs
OUT_DIM_64 = {2: 29, 4: 25, 6: 21}

class PixelEncoder(nn.Module):
    """Convolutional encoder of pixels observations."""
    def __init__(self, obs_shape, feature_dim, num_layers=2,
                  num_filters=32, output_logits=False):
        super(PixelEncoder, self).__init__()

        assert len(obs_shape) == 3
        self.obs_shape = obs_shape
        self.feature_dim = feature_dim
        self.num_layers = num_layers

        self.convs = nn.ModuleList(
            [nn.Conv2d(obs_shape[0], num_filters, 3, stride=2)
             ]
        )
        for i in range(num_layers - 1):
            self.convs.append(nn.Conv2d(num_filters,
                                         num_filters, 3, stride=1))

        out_dim = OUT_DIM_64[num_layers] if obs_shape[-1] ==
            64 else OUT_DIM[num_layers]
        self.fc = nn.Linear(num_filters * out_dim * out_dim,
                             self.feature_dim)
        self.ln = nn.LayerNorm(self.feature_dim)

        self.outputs = dict()
        self.output_logits = output_logits

    def reparameterize(self, mu, logstd):
        std = torch.exp(logstd)
        eps = torch.randn_like(std)
        return mu + eps * std

    def forward_conv(self, obs):

```

```

obs = obs / 255.
self.outputs['obs'] = obs

conv = torch.relu(self.convs[0](obs))
self.outputs['conv1'] = conv

for i in range(1, self.num_layers):
    conv = torch.relu(self.convs[i](conv))
    self.outputs['conv%s' % (i + 1)] = conv

h = conv.view(conv.size(0), -1)
return h

def forward(self, obs, detach=False):
    h = self.forward_conv(obs)

    if detach:
        h = h.detach()

    h_fc = self.fc(h)
    self.outputs['fc'] = h_fc

    h_norm = self.ln(h_fc)
    self.outputs['ln'] = h_norm

    if self.output_logits:
        out = h_norm
    else:
        out = torch.tanh(h_norm)
        self.outputs['tanh'] = out

    return out

def copy_conv_weights_from(self, source):
    """Tie convolutional layers"""
    # only tie conv layers
    for i in range(self.num_layers):
        tie_weights(src=source.convs[i], trg=self.convs[i]

```

```

# def log(self, L, step, log_freq):
# if step % log_freq != 0:
# return

# for k, v in self.outputs.items():
# L.log_histogram('train_encoder/%s_hist' % k, v, step)
# if len(v.shape) > 2:
# L.log_image('train_encoder/%s_img' % k, v[0], step)

# for i in range(self.num_layers):
# L.log_param('train_encoder/conv%s' % (i + 1), self.
#     convs[i], step)
# L.log_param('train_encoder/fc', self.fc, step)
# L.log_param('train_encoder/ln', self.ln, step)

class IdentityEncoder(nn.Module):
    def __init__(self, obs_shape, feature_dim, num_layers,
        num_filters, *args):
        super().__init__()

        assert len(obs_shape) == 1
        self.feature_dim = obs_shape[0]

    def forward(self, obs, detach=False):
        return obs

    def copy_conv_weights_from(self, source):
        pass

# def log(self, L, step, log_freq):
# pass

_AVAILABLE_ENCODERS = {'pixel': PixelEncoder, 'identity':
    IdentityEncoder}

```

```

def make_encoder(
    encoder_type, obs_shape, feature_dim, num_layers,
    num_filters, output_logits=False
):
    assert encoder_type in _AVAILABLE_ENCODERS
    return _AVAILABLE_ENCODERS[encoder_type](
        obs_shape, feature_dim, num_layers, num_filters,
        output_logits
    )

LOG_FREQ = 10000

def gaussian_logprob(noise, log_std):
    """Compute Gaussian log probability."""
    residual = (-0.5 * noise.pow(2) - log_std).sum(-1,
        keepdim=True)
    return residual - 0.5 * np.log(2 * np.pi) * noise.size
    (-1)

def squash(mu, pi, log_pi):
    """Apply squashing function.
    See appendix C from https://arxiv.org/pdf/1812.05905.pdf
    .
    """
    mu = torch.tanh(mu)
    if pi is not None:
        pi = torch.tanh(pi)
    if log_pi is not None:
        log_pi -= torch.log(F.relu(1 - pi.pow(2)) + 1e-6).sum
            (-1, keepdim=True)
    return mu, pi, log_pi

def weight_init(m):
    """Custom weight init for Conv2D and Linear layers."""
    if isinstance(m, nn.Linear):
        nn.init.orthogonal_(m.weight.data)

```

```

        m.bias.data.fill_(0.0)
elif isinstance(m, nn.Conv2d) or isinstance(m, nn.
ConvTranspose2d):
    # delta-orthogonal init from https://arxiv.org/pdf
    /1806.05393.pdf
    assert m.weight.size(2) == m.weight.size(3)
    m.weight.data.fill_(0.0)
    m.bias.data.fill_(0.0)
    mid = m.weight.size(2) // 2
    gain = nn.init.calculate_gain('relu')
    nn.init.orthogonal_(m.weight.data[:, :, mid, mid],
        gain)

class Actor(nn.Module):
    """MLP actor network."""
    def __init__(
        self, obs_shape, action_shape, hidden_dim,
            encoder_type,
        encoder_feature_dim, log_std_min, log_std_max,
            num_layers, num_filters
    ):
        super(Actor, self).__init__()

        self.encoder = make_encoder(
            encoder_type, obs_shape, encoder_feature_dim,
                num_layers,
            num_filters, output_logits=True
        )

        self.log_std_min = log_std_min
        self.log_std_max = log_std_max

        self.trunk = nn.Sequential(
            nn.Linear(self.encoder.feature_dim + 3*6,
                hidden_dim), nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim), nn.ReLU(),
            nn.Linear(hidden_dim, 2 * action_shape)
        )

```

```

self.outputs = dict()
self.apply(weight_init)

def forward(
    self, obs, pose, compute_pi=True, compute_log_pi=True
        , detach_encoder=False
):
    obs = self.encoder(obs, detach=detach_encoder)

    # print('encoder_q-----')
    # # print(obs)
    # # print(type(obs))
    # # print(obs.size())
    # print(pose)
    # print(type(pose))
    # print(pose.size())
    obs_pose = torch.cat([obs, pose], dim=1)
    # print(type(obs_pose))
    # print('-----')

    mu, log_std = self.trunk(obs_pose).chunk(2, dim=-1)

    # constrain log_std inside [log_std_min, log_std_max]
    log_std = torch.tanh(log_std)
    log_std = self.log_std_min + 0.5 * (
        self.log_std_max - self.log_std_min
    ) * (log_std + 1)

    self.outputs['mu'] = mu
    self.outputs['std'] = log_std.exp()

    if compute_pi:
        std = log_std.exp()
        noise = torch.randn_like(mu)
        pi = mu + noise * std
    else:
        pi = None
        entropy = None

```

```

    if compute_log_pi:
        log_pi = gaussian_logprob(noise, log_std)
    else:
        log_pi = None

    mu, pi, log_pi = squash(mu, pi, log_pi)

    return mu, pi, log_pi, log_std

# def log(self, L, step, log_freq=LOG_FREQ):
# if step % log_freq != 0:
# return

# for k, v in self.outputs.items():
# L.log_histogram('train_actor/%s_hist' % k, v, step)

# L.log_param('train_actor/fc1', self.trunk[0], step)
# L.log_param('train_actor/fc2', self.trunk[2], step)
# L.log_param('train_actor/fc3', self.trunk[4], step)

class QFunction(nn.Module):
    """MLP for q-function."""
    def __init__(self, obs_dim, action_dim, hidden_dim):
        super(QFunction, self).__init__()

        self.trunk = nn.Sequential(
            nn.Linear(obs_dim + 3+6 + action_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim), nn.ReLU(),
            nn.Linear(hidden_dim, 1)
        )

    def forward(self, obs, pose, action):
        assert obs.size(0) == action.size(0)

        obs_pose = torch.cat([obs, pose], dim=1)
        obs_action = torch.cat([obs_pose, action], dim=1)

```

```

        return self.trunk(obs_action)

class Critic(nn.Module):
    """Critic network, employs two q-functions."""
    def __init__(
        self, obs_shape, action_shape, hidden_dim,
            encoder_type,
            encoder_feature_dim, num_layers, num_filters
    ):
        super(Critic, self).__init__()

        self.encoder = make_encoder(
            encoder_type, obs_shape, encoder_feature_dim,
            num_layers,
            num_filters, output_logits=True
        )

        self.Q1 = QFunction(
            self.encoder.feature_dim, action_shape, hidden_dim
        )
        self.Q2 = QFunction(
            self.encoder.feature_dim, action_shape, hidden_dim
        )

        self.outputs = dict()
        self.apply(weight_init)

    def forward(self, obs, pose, action, detach_encoder=
        False):
        # detach_encoder allows to stop gradient propogation
        to encoder
        obs = self.encoder(obs, detach=detach_encoder)

        # print('aquil13', obs.shape)

        q1 = self.Q1(obs, pose, action)
        q2 = self.Q2(obs, pose, action)

```

```

        self.outputs['q1'] = q1
        self.outputs['q2'] = q2

    return q1, q2

# def log(self, L, step, log_freq=LOG_FREQ):
# if step % log_freq != 0:
# return

# self.encoder.log(L, step, log_freq)

# for k, v in self.outputs.items():
# L.log_histogram('train_critic/%s_hist' % k, v, step)

# for i in range(3):
# L.log_param('train_critic/q1_fc%d' % i, self.Q1.trunk[
#     i * 2], step)
# L.log_param('train_critic/q2_fc%d' % i, self.Q2.trunk[
#     i * 2], step)

class Depth_CUPRL(nn.Module):
    """
    Depth_CUPRL
    """

    def __init__(self, obs_shape, z_dim, batch_size, critic,
        critic_target, output_type="continuous"):
        super(Depth_CUPRL, self).__init__()
        self.batch_size = batch_size

        self.encoder = critic.encoder

        self.encoder_target = critic_target.encoder

        self.W = nn.Parameter(torch.rand(z_dim, z_dim))
        self.output_type = output_type

```

```

def encode(self, x, detach=False, ema=False):
    """
    Encoder:  $z_t = e(x_t)$ 
    :param x:  $x_t$ , x y coordinates
    :return:  $z_t$ , value in  $r_2$ 
    """
    if ema:
        with torch.no_grad():
            z_out = self.encoder_target(x)
    else:
        z_out = self.encoder(x)

    if detach:
        z_out = z_out.detach()
    return z_out

def compute_logits(self, z_a, z_pos):
    """
    Uses logits trick for Depth_CUPRL:
    - compute  $(B, B)$  matrix  $z_a$  ( $W z_{pos}.T$ )
    - positives are all diagonal elements
    - negatives are all other elements
    - to compute loss use multiclass cross entropy with
      identity matrix for labels
    """
    Wz = torch.matmul(self.W, z_pos.T) #  $(z_{dim}, B)$ 
    logits = torch.matmul(z_a, Wz) #  $(B, B)$ 
    logits = logits - torch.max(logits, 1)[0][:, None]
    return logits

class CurlSacAgent(object):
    """Depth_CUPRL representation learning with SAC."""
    def __init__(
        self,
        obs_shape,
        action_shape,
        device,
        hidden_dim=256,
        discount=0.99,
    ):

```

```

init_temperature=0.01,
alpha_lr=1e-3,
alpha_beta=0.9,
actor_lr=1e-3,
actor_beta=0.9,
actor_log_std_min=-10,
actor_log_std_max=2,
actor_update_freq=2,
critic_lr=1e-3,
critic_beta=0.9,
critic_tau=0.005,
critic_target_update_freq=2,
encoder_type='pixel',
encoder_feature_dim=50,
encoder_lr=1e-3,
encoder_tau=0.005,
num_layers=4,
num_filters=32,
cpc_update_freq=1,
log_interval=100,
detach_encoder=False,
curl_latent_dim=128
):
    self.device = device
    self.discount = discount
    self.critic_tau = critic_tau
    self.encoder_tau = encoder_tau
    self.actor_update_freq = actor_update_freq
    self.critic_target_update_freq =
        critic_target_update_freq
    self.cpc_update_freq = cpc_update_freq
    self.log_interval = log_interval
    self.image_size = obs_shape[-1]
    self.curl_latent_dim = curl_latent_dim
    self.detach_encoder = detach_encoder
    self.encoder_type = encoder_type

    self.actor = Actor(
        obs_shape, action_shape, hidden_dim, encoder_type,

```

```

        encoder_feature_dim, actor_log_std_min,
        actor_log_std_max,
        num_layers, num_filters
    ).to(device)

    self.critic = Critic(
        obs_shape, action_shape, hidden_dim, encoder_type,
        encoder_feature_dim, num_layers, num_filters
    ).to(device)

    self.critic_target = Critic(
        obs_shape, action_shape, hidden_dim, encoder_type,
        encoder_feature_dim, num_layers, num_filters
    ).to(device)

    self.critic_target.load_state_dict(self.critic.
        state_dict())

    # tie encoders between actor and critic, and
    # Depth_CUPRL and critic
    self.actor.encoder.copy_conv_weights_from(self.critic
        .encoder)

    self.log_alpha = torch.tensor(np.log(init_temperature
        )).to(device)
    self.log_alpha.requires_grad = True
    # set target entropy to  $-|A|$ 
    self.target_entropy = -np.prod(action_shape)

    # optimizers
    self.actor_optimizer = torch.optim.Adam(
        self.actor.parameters(), lr=actor_lr, betas=(
            actor_beta, 0.999)
    )

    self.critic_optimizer = torch.optim.Adam(
        self.critic.parameters(), lr=critic_lr, betas=(
            critic_beta, 0.999)
    )

```

```

self.log_alpha_optimizer = torch.optim.Adam(
    [self.log_alpha], lr=alpha_lr, betas=(alpha_beta,
        0.999)
)

if self.encoder_type == 'pixel':
    # create Depth_CUPRL encoder (the 128 batch size
    # is probably unnecessary)
    self.Depth_CUPRL = Depth_CUPRL(obs_shape,
        encoder_feature_dim,
        self.curl_latent_dim, self.critic, self.
            critic_target, output_type='continuous
            ').to(self.device)

    # optimizer for critic encoder for reconstruction
    loss
    self.encoder_optimizer = torch.optim.Adam(
        self.critic.encoder.parameters(), lr=encoder_lr
    )

    self.cpc_optimizer = torch.optim.Adam(
        self.Depth_CUPRL.parameters(), lr=encoder_lr
    )
self.cross_entropy_loss = nn.CrossEntropyLoss()

self.train()
self.critic_target.train()

def train(self, training=True):
    self.training = training
    self.actor.train(training)
    self.critic.train(training)
    if self.encoder_type == 'pixel':
        self.Depth_CUPRL.train(training)

@property
def alpha(self):
    return self.log_alpha.exp()

```

```

def select_action(self, obs, pose):
    if obs.shape[-1] != self.image_size:
        obs = center_crop_image(obs, self.image_size)

    with torch.no_grad():
        obs = torch.FloatTensor(obs).to(self.device)
        pose = torch.FloatTensor([pose]).to(self.device)
        obs = obs.unsqueeze(0)
        mu, _, _, _ = self.actor(obs, pose, compute_log_pi
                                =False)
        return mu.cpu().data.numpy().flatten()

def sample_action(self, obs, pose):
    if obs.shape[-1] != self.image_size:
        obs = center_crop_image(obs, self.image_size)

    with torch.no_grad():
        obs = torch.FloatTensor(obs).to(self.device)
        pose = torch.FloatTensor([pose]).to(self.device)
        obs = obs.unsqueeze(0)
        mu, pi, _, _ = self.actor(obs, pose,
                                compute_log_pi=False)
        return pi.cpu().data.numpy().flatten()

def update_critic(self, obs, pose, action, reward,
next_obs, next_pose, not_done, step, weights, writer)
:
    weights = torch.FloatTensor(weights).unsqueeze(1).to(
        self.device)

    with torch.no_grad():
        _, policy_action, log_pi, _ = self.actor(next_obs,
            next_pose)
        target_Q1, target_Q2 = self.critic_target(next_obs
            , next_pose, policy_action)
        target_V = torch.min(target_Q1,
                                target_Q2) - self.alpha.detach() *
                                log_pi

```

```

        target_Q = reward + (not_done * self.discount *
                               target_V)

    # get current Q estimates
    current_Q1, current_Q2 = self.critic(
        obs, pose, action, detach_encoder=self.
        detach_encoder)
    # critic_loss = F.mse_loss(current_Q1,
    # target_Q) + F.mse_loss(current_Q2, target_Q)
    td_error1 = target_Q.detach()-current_Q1
    td_error2 = target_Q.detach()-current_Q2
    critic1_loss = 0.5* (td_error1.pow(2)*weights).mean()
    critic2_loss = 0.5* (td_error2.pow(2).to(self.device)
        *weights).mean()
    prios = abs((td_error1 + td_error2)/2.0 + 1e-5).
        squeeze())

    critic_loss = critic1_loss + critic2_loss

    writer.add_scalar('Loss_critic', critic_loss, step)
    # if step % self.log_interval == 0:
    # L.log('train_critic/loss', critic_loss, step)

    # Optimize the critic
    self.critic_optimizer.zero_grad()
    critic_loss.backward()
    self.critic_optimizer.step()

    # self.critic.log(L, step)
    return prios

def update_actor_and_alpha(self, obs, pose, step):
    # detach encoder, so we don't update it with the
    actor loss
    _, pi, log_pi, log_std = self.actor(obs, pose,
        detach_encoder=True)
    actor_Q1, actor_Q2 = self.critic(obs, pose, pi,
        detach_encoder=True)

```

```

actor_Q = torch.min(actor_Q1, actor_Q2)
actor_loss = (self.alpha.detach() * log_pi - actor_Q)
               .mean()

# if step % self.log_interval == 0:
#     L.log('train_actor/loss', actor_loss, step)
#     L.log('train_actor/target_entropy', self.
#           target_entropy, step)
entropy = 0.5 * log_std.shape[1] * \
          (1.0 + np.log(2 * np.pi)) + log_std.sum(dim=-1)
# if step % self.log_interval == 0:
#     L.log('train_actor/entropy', entropy.mean(),
#           step)

# optimize the actor
writer.add_scalar('Loss_actor', actor_loss, step)

self.actor_optimizer.zero_grad()
actor_loss.backward()
self.actor_optimizer.step()

# self.actor.log(L, step)

self.log_alpha_optimizer.zero_grad()
alpha_loss = (self.alpha *
              (-log_pi - self.target_entropy).detach()).
              mean()

writer.add_scalar('Loss_alpha', alpha_loss, step)
# if step % self.log_interval == 0:
#     L.log('train_alpha/loss', alpha_loss, step)
#     L.log('train_alpha/value', self.alpha, step)
alpha_loss.backward()
self.log_alpha_optimizer.step()

def update_cpc(self, obs_anchor, obs_pos, cpc_kwargs,
               step):

```

```

z_a = self.Depth_CUPRL.encode(obs_anchor)
z_pos = self.Depth_CUPRL.encode(obs_pos, ema=True)

logits = self.Depth_CUPRL.compute_logits(z_a, z_pos)
labels = torch.arange(logits.shape[0]).long().to(self
    .device)
loss = self.cross_entropy_loss(logits, labels)

self.encoder_optimizer.zero_grad()
self.cpc_optimizer.zero_grad()
loss.backward()

writer.add_scalar('Loss_curl', loss, step)

self.encoder_optimizer.step()
self.cpc_optimizer.step()
# if step % self.log_interval == 0:
#     L.log('train/curl_loss', loss, step)

def update(self, replay_buffer, step, writer):
    if self.encoder_type == 'pixel':
        obs, pose, action, reward, next_obs, next_pose,
            not_done, cpc_kwargs, idx, weights =
            replay_buffer.sample_cpc()
    else:
        obs, action, reward, next_obs, not_done =
            replay_buffer.sample_proprio()

    # if step % self.log_interval == 0:
    #     L.log('train/batch_reward', reward.mean(), step)

    prios = self.update_critic(obs, pose, action, reward,
        next_obs, next_pose, not_done, step, weights,
        writer)

    replay_buffer.update_priorities(idx, prios.data.cpu()
        .numpy())

```

```

if step % self.actor_update_freq == 0:
    self.update_actor_and_alpha(obs, pose, step)

if step % self.critic_target_update_freq == 0:
    soft_update_params(
        self.critic.Q1, self.critic_target.Q1, self.
        critic_tau
    )
    soft_update_params(
        self.critic.Q2, self.critic_target.Q2, self.
        critic_tau
    )
    soft_update_params(
        self.critic.encoder, self.critic_target.encoder
        ,
        self.encoder_tau
    )

if step % self.cpc_update_freq == 0 and self.
encoder_type == 'pixel':
    obs_anchor, obs_pos = cpc_kwargs["obs_anchor"],
        cpc_kwargs["obs_pos"]
    self.update_cpc(obs_anchor, obs_pos, cpc_kwargs,
        step)

def save(self, model_dir, step):
    torch.save(
        self.actor.state_dict(), '%s/SAC_model/actor_%s.pt'
        ' % (model_dir, step)
    )
    torch.save(
        self.critic.state_dict(), '%s/SAC_model/critic_%s.
        pt' % (model_dir, step)
    )

def save_curl(self, model_dir, step):
    torch.save(
        self.Depth_CUPRL.state_dict(), '%s/SAC_model/curl_
        %s.pt' % (model_dir, step)

```

```

    )

    def load(self, model_dir, step):
        self.actor.load_state_dict(
            torch.load('%s/SAC_model/actor_%s.pt' % (model_dir,
                , step))
        )
        self.critic.load_state_dict(
            torch.load('%s/SAC_model/critic_%s.pt' % (
                model_dir, step))
        )
        self.Depth_CUPRL.load_state_dict(
            torch.load('%s/SAC_model/curl_%s.pt' % (model_dir,
                step))
        )

    def make_agent(obs_shape, action_shape, device):
        # if args.agent == 'curl_sac':
        return CurlSacAgent(
            obs_shape=obs_shape,
            action_shape=action_shape,
            device=device,
            hidden_dim=1024,
            discount=0.99,
            init_temperature=0.1,
            alpha_lr=1e-4,
            alpha_beta=0.5,
            actor_lr=1e-3,
            actor_beta=0.9,
            actor_log_std_min=-10,
            actor_log_std_max=2,
            actor_update_freq=2,
            critic_lr=1e-3,
            critic_beta=0.9,
            critic_tau=0.01,
            critic_target_update_freq=2,
            encoder_type='pixel',
            encoder_feature_dim=50,
            encoder_lr=1e-3,

```

```

        encoder_tau=0.05,
        num_layers=4,
        num_filters=32,
        log_interval=100,
        detach_encoder=False,
        curl_latent_dim=128

    )
    # else:
    # assert 'agent is not supported: %s' % args.agent

#-----
ACTION_V_MIN = -0.3 # m/s
ACTION_W_MIN = -0.3 # rad/s
ACTION_V_MAX = 0.3 # m/s
ACTION_W_MAX = 0.3 # rad/s
replay_buffer_size = 100000
#*****
is_training = True
#-----
def action_unnormalized(action, high, low):
    action = low + (action + 1.0) * 0.5 * (high - low)
    action = np.clip(action, low, high)
    return action
#*****

writer = SummaryWriter(dirPath + '/evaluations/', flush_secs
    =1, max_queue=1)

def evaluate(num_episodes=3, encoder_type='pixel',
    image_size=84):
    all_ep_rewards = []
    print('evau')

    def run_eval_loop(sample_stochastically=True):

        prefix = 'stochastic_' if sample_stochastically else
            ''

```

```

if sample_stochastically:
    print('stochastically_evaluation')
else:
    print('deterministic_evaluation')

for i in range(num_episodes):
    obs, pose = env.reset()
    pose = np.array(pose)
    # print('-print aqui-----')
    done = False
    episode_reward = 0
    step = 0
    while not done:
        if encoder_type == 'pixel':
            obs = center_crop_image(obs, image_size)
        with eval_mode(agent):
            if sample_stochastically:
                action = agent.sample_action(obs, pose)
                unnorm_action = np.array([
                    action_unnormalized(action[0],
                                         ACTION_V_MAX, 0),
                    action_unnormalized(action[1],
                                         ACTION_W_MAX, ACTION_W_MIN),
                    action_unnormalized(action[2],
                                         ACTION_V_MAX, ACTION_V_MIN)
                ])
            else:
                action = agent.select_action(obs, pose)
                unnorm_action = np.array([
                    action_unnormalized(action[0],
                                         ACTION_V_MAX, 0),
                    action_unnormalized(action[1],
                                         ACTION_W_MAX, ACTION_W_MIN),
                    action_unnormalized(action[2],
                                         ACTION_V_MAX, ACTION_V_MIN)
                ])
            obs, pose, reward, done = env.step(
                unnorm_action)

```

```

        pose = np.array(pose)
        # print('pose', pose)

        episode_reward += reward
        step +=1
        if step > 1000:
            print('final', i, step)
            done = True
            break

    all_ep_rewards.append(episode_reward)

    mean_ep_reward = np.mean(all_ep_rewards)
    best_ep_reward = np.max(all_ep_rewards)

    print('Evaluation: _Mean, _Best')
    print(mean_ep_reward, best_ep_reward)

    return mean_ep_reward, best_ep_reward

return run_eval_loop(sample_stochastically=False)

if __name__ == '__main__':
    rospy.init_node('curl_node')
    pub_result = rospy.Publisher('result', Float32,
        queue_size=5)
    result = Float32()
    env = Env()

    action_shape = 3
    obs_shape = (3*3, 84, 84)
    pre_aug_obs_shape = (3*3, 100, 100)

    episode, episode_reward, done = 0, 0, True
    max_steps = 1000000
    initial_step = 0
    # initial_step = 25000
    save_model_replay = False

```

```

device = torch.device('cuda' if torch.cuda.is_available
    () else 'cpu')

replay_buffer = ReplayBuffer(
    obs_shape=pre_aug_obs_shape,
    action_shape=action_shape,
    capacity=replay_buffer_size,
    batch_size=128,
    device=device,
    image_size=84,
)

agent = make_agent(
    obs_shape=obs_shape,
    action_shape=action_shape,
    device=device
)

print('Path:_' + dirPath)

# agent.load(dirPath, initial_step) # 5 + 149 + 30
# replay_buffer.load(dirPath + '/replay_memory/')

for step in range(initial_step, max_steps):

    if step % 1000 == 0:
        print('start_eval')
        mean, best = evaluate()
        writer.add_scalar('Reward_mean', mean, step)
        writer.add_scalar('Reward_best', best, step)
        if save_model_replay:
            if step%(1*1000) == 0:
                agent.save(dirPath, step)
                agent.save_curl(dirPath, step)
                # replay_buffer.save(dirPath + '/'
                    replay_memory/')
            print('saved_model_and_replay_memory', step)
        # obs = env.reset()
        done = True

```

```

save_model_replay = True

if done:

    print("*****")
    print('Episode:_' + str(episode) + '_training')
    print('Step:_' + str(step) + '_training')
    print('Reward_average_per_ep:_' + str(
        episode_reward))
    print("*****")

    obs, pose = env.reset()
    pose = np.array(pose)

    done = False
    episode_reward = 0
    episode += 1

if step < 200: #1000
    print('###_Collecting_memory_###')
    action = np.array([
        np.random.uniform(-1, 1),
        np.random.uniform(-1, 1),
        np.random.uniform(-1, 1)
    ])
    unnorm_action = np.array([
        action_unnormalized(action[0], ACTION_V_MAX, 0)
        ,
        action_unnormalized(action[1], ACTION_W_MAX,
            ACTION_W_MIN),
        action_unnormalized(action[2], ACTION_V_MAX,
            ACTION_V_MIN)
    ])
else:
    with eval_mode(agent):
        action = agent.sample_action(obs, pose)
        # print('action1 ', unnorm_action)
        unnorm_action = np.array([
            action_unnormalized(action[0], ACTION_V_MAX,

```

```

        0),
        action_unnormalized(action[1], ACTION_W_MAX,
                             ACTION_W_MIN),
        action_unnormalized(action[2], ACTION_V_MAX,
                             ACTION_V_MIN)
    ])

if step >= (initial_step + 400): #1000
    num_updates = 1
    for _ in range(num_updates):
        # print('update')
        agent.update(replay_buffer, step, writer)

next_obs, next_pose, reward, done = env.step(
    unnorm_action)
next_pose = np.array(next_pose)
# print('----')
# print('action ', unnorm_action)
# print('pose ', next_pose)
# print('reward ', reward)
# print('----')
# print('state ', obs.shape)

episode_reward += reward
replay_buffer.add(obs, pose, action, reward, next_obs
    , next_pose, done)
if reward <= -1.:
    print('\\n----collide-----\\n')
    # for _ in range(1):
    # # print('aqui2')
    # replay_buffer.add(obs, action, reward, next_obs,
        done)

obs = copy.deepcopy(next_obs)
pose = copy.deepcopy(next_pose)
# episode_step += 1

```