Universidade Federal do Rio Grande do Norte
Centro de Ciências Exatas e da Terra
Departamento de Informática e Matemática Aplicada
Bacharelado em Ciência da Computação

Comparação de desempenho das implementações do algoritmo *Deep Q-Network* em *Rust* e *Python* no ambiente *Cart Pole*

João Vítor Venceslau Coelho

Natal-RN
Agosto 2024

João Vítor Venceslau Coelho

Comparação de desempenho das implementações do algoritmo *Deep Q-Network* em *Rust* e *Python* no ambiente *Cart Pole*

Monografia de Graduação apresentada ao Departamento de Informática e Matemática Aplicada do Centro de Ciências Exatas e da Terra da Universidade Federal do Rio Grande do Norte como requisito parcial para a obtenção do grau de bacharel em Ciência da Computação.

Orientador

Professor Doutor Charles Andryê Galvão Madeira

Co-orientador

Professor Doutor Samuel Xavier de Souza

DIMAp – Departamento de Informática e Matemática Aplicada
CCET – Centro de Ciências Exatas e da Terra
UFRN – Universidade Federal do Rio Grande do Norte

Natal-RN

Agosto 2024

Universidade Federal do Rio Grande do Norte - UFRN
Sistema de Bibliotecas - SISBI
Catalogação de Publicação na Fonte. UFRN - Biblioteca Setorial Prof. Ronaldo Xavier de Arruda - CCET

Coelho, João Vitor Venceslau.
Comparação de desempenho das implementações do algoritmo Deep Q-Network em Rust e Python no ambiente Cart Pole / João Vitor Venceslau Coelho. - 2024.
64 f.: il.

Monografia (graduação) - Universidade Federal do Rio Grande do Norte, Centro de Ciências Exatas e da Terra, Curso de Ciência da Computação. Natal, RN, 2024.
Orientação: Prof. Dr. Charles Andryê Galvão Madeira.
Coorientação: Prof. Dr. Samuel Xavier de Souza.

1. Computação - Monografia. 2. Aprendizado por reforço profundo - Monografia. 3. Rust - Monografia. 4. Deep Q-Network - Monografia. 5. Eficiência computacional - Monografia. 6. Comparação de desempenho - Monografia. I. Madeira, Charles Andryê Galvão. II. Souza, Samuel Xavier de. III. Título.

RN/UF/CCET CDU 004

Elaborado por Joseneide Ferreira Dantas - CRB-15/324

Monografia de Graduação sob o título *Comparação de desempenho das implementações do algoritmo Deep Q-Network em Rust e Python no ambiente Cart Pole* apresentada por João Vítor Venceslau Coelho e aceita pelo Departamento de Informática e Matemática Aplicada do Centro de Ciências Exatas e da Terra da Universidade Federal do Rio Grande do Norte, sendo aprovada por todos os membros da banca examinadora abaixo especificada:

Professor Doutor Charles Andryê Galvão Madeira
Orientador(a)
Instituto Metrópole Digital
Universidade Federal do Rio Grande do Norte

Professor Doutor Samuel Xavier de Souza
Co-orientador(a)
Departamento de Engenharia de Computação e Automação
Universidade Federal do Rio Grande do Norte

Professor Doutor Ranniery da Silva Maia
Departamento de Informática e Matemática Aplicada
Universidade Federal do Rio Grande do Norte

Professor Doutor Daniel Sabino Amorim de Araujo
Instituto Metrópole Digital
Universidade Federal do Rio Grande do Norte

Natal-RN, 16 de agosto de 2024.

Pelo suporte, paciência, e dedicação a me ajudar na escrita desta monografia e sobretudo, pela grande companheira que é, dedico esta monografia a minha amada Sabrina. Não tenho palavras para expressar o quanto sou grato por lhe ter ao meu lado.

Agradecimentos

Agradeço a todas as pessoas que me ajudaram de alguma maneira a chegar a esse momento. Agradeço minha mãe e meu pai por todos os incentivos a estudar e por proporcionarem as condições para que eu pudesse me dedicar aos estudos por tantos anos. Agradeço aos diversos amigos que fiz durante o curso, aproveitei muito a companhia de todos. Agradeço aos diversos professores, por todos os ensinamentos nas mais variadas disciplinas. A UFRN como um todo por propiciar o curso e o ambiente onde estudei durante tantos anos. Por fim, agradeço minha amada Sabrina, por tanto me ajudar em diversos momentos na escrita dessa monografia e por todo o apoio que sempre me deu.

Between stimulus and response, there is a space.
In that space is our power to choose our response.
In our response lies our growth and our freedom.

Viktor Frankl

Comparação de desempenho das implementações do algoritmo *Deep Q-Network* em *Rust* e *Python* no ambiente *Cart Pole*

Autor: João Vítor Venceslau Coelho
Orientador(a): Professor Doutor Charles Andryê Galvão Madeira
Co-orientador(a): Professor Doutor Samuel Xavier de Souza

Resumo

O aprendizado por reforço profundo ganhou destaque devido ao seu sucesso na resolução de problemas complexos, mas seus altos tempos de execução aumentam os custos, muitas vezes necessitando de hardware poderoso e caro. Este estudo visa reduzir esses custos aumentando a eficiência de algoritmos de aprendizagem por reforço profundo utilizando a linguagem *Rust*, conhecida por sua geração eficiente de código e gerenciamento seguro de memória sem coletor de lixo. Comparamos uma implementação *Rust* do algoritmo Deep Q-Network (DQN) com a implementação *Python* da biblioteca *Stable Baselines3* no ambiente *Cart Pole*. Os resultados mostram que a implementação do *Rust* é mais rápida, mesmo com *bindings* para *Python*, alcançando uma menor proporção de tempo por passo.

Palavras-chave: Aprendizado por Reforço Profundo, Rust, Deep Q-Network, Eficiência Computacional, Comparação de Desempenho

Performance comparison of implementations of the Deep Q-Network algorithm in Rust and Python in the Cart Pole environment

Author: João Vítor Venceslau Coelho
Supervisor: Professor Doctor Charles Andryê Galvão Madeira
Co-supervisor(a): Professor Doctor Samuel Xavier de Souza

Abstract

Deep reinforcement learning has gained prominence due to its success in solving complex problems, but its high execution times increase costs, often requiring powerful and expensive hardware. This study aims to reduce these costs by enhancing the efficiency of deep reinforcement learning algorithms using the Rust language, known for its efficient code generation and safe memory management without a garbage collector. We compare a Rust implementation of the Deep Q-Network (DQN) algorithm with the Python implementation from the Stable Baselines3 library in the Cart Pole environment. Results show that the Rust implementation is faster, even with Python bindings, achieving a lower time per step ratio.

Keywords: Deep Reinforcement Learning, Rust, Deep Q-Network (DQN), Computational Efficiency, Performance Comparison

Lista de figuras

Lista de tabelas

Lista de abreviaturas e siglas

DRL – deep reinforcement learning

GPU – graphics processing unit

RL – Reinforcement Learning

DQN – Deep Q-Network

MLP – Multilayer Perceptron

RBF – Radial Basis Function

MDP – Markov Decicion Process

CPU – Central Process Unit

TPU – Tensor Process Unit

API – Application Program Interfaces

SB3 – Stable Baselines3

CLI – Command Line Interface

TD3 – Twin Delayed Deep Deterministic

PPO – Proximal Policy Optimization

SAC – Soft Actor-Critic

Lista de símbolos

S (Conjunto de estados de um MDP)

A (Conjunto de ações em um MDP)

A_s (Conjunto de ações permitidas no estado s de um MDP)

P (Conjunto de probabilidades de transição entre estados de um MDP após cada ação)

s_t (Estado atual no momento t)

s'_t (Estado seguinte após uma ação no momento t)

$P_a(s_t, s'_t)$ (Elementos de P)

R (Conjunto de recompensas de um MDP por transição entre estados após cada ação)

$R_a(s_t, s'_t)$ (Elementos de R)

π (Uma política de um agente)

Sumário

1 Introdução

Neste primeiro momento é introduzida uma contextualização geral da área de aprendizado por reforço profundo, *deep reinforcement learning* (*DRL*) em inglês, sendo apresentada sua relevância nos dias atuais e o problema abordado. Em seguida, é explicada a motivação e a justificativa do trabalho realizado, bem como os objetivos a serem alcançados e, por fim, a estrutura do texto é apresentada.

1.1 Contextualização e problema abordado

As técnicas de aprendizado por reforço profundo ganharam destaque nos últimos 10 anos devido aos avanços no uso de redes neurais profundas (LECUN; BENGIO; HINTON, 2015; GOODFELLOW; BENGIO; COURVILLE, 2016).

Essas técnicas já alcançaram feitos impressionantes, como o *AlphaGo* (SILVER et al., 2017), o *DeepStack* (MORAVčíK et al., 2017) e, recentemente, também houve melhorias nos algoritmos de multiplicação de matrizes e ordenação (FAWZI et al., 2022; MANKOWITZ et al., 2023). Porém, apesar dos avanços em escalabilidade e eficiência de modelos de *DRL*, especialmente com o uso das *Graphics Processing Units* ou *GPUs* (LIANG et al., 2018; RUDIN et al., 2022), ainda existe um grande desafio em lidar com a necessidade de interação extensiva com o ambiente, especialmente em ambientes de alta complexidade.

Atualmente, a grande maioria dos *frameworks* disponíveis para abordar problemas de aprendizado de máquina são desenvolvidos em *Python*, não sendo diferente para as bibliotecas de aprendizado por reforço profundo, podendo citar o *TensorFlow Agents*, o *TensorForce* e o *Reinforcement Learning Coach* da Intel, que são baseados no *TensorFlow*, como também o *TorchRL* e o *MushroomRL* à base do *PyTorch*. Além disso, no que se refere ao uso das redes neurais, todas essas bibliotecas, por se basearem em bibliotecas consolidadas para uso de redes neurais, já são devidamente otimizadas, o que acelera

consideravelmente o treinamento de novos modelos.

Entretanto, a implementação dos algoritmos em uma linguagem interpretada como o *Python* aumenta os custos da utilização de laços de repetição, de fluxo de controle, de conversões e de outras diversas operações. Salientando ainda que, com o uso que os algoritmos envolvidos fazem das redes neurais, ainda há os custos das chamadas aos *bindings* das bibliotecas relacionadas, como o *TensorFlow* ou *PyTorch*, que movem os dados entre *C/C++* e *Python*, como também para a *GPU*.

Isso posto, devido a interação do agente com o ambiente, a comunicação entre o *Python* e as bibliotecas de redes neurais acontece de forma intensiva. Porém, mesmo com a diminuição dessa comunicação, devido ao uso de aprendizado em lote, ainda assim não é o suficiente, criando um *overhead* a longo prazo, o que aumenta o tempo de execução.

1.2 Motivação e Justificativa

Diante do desafio exposto, uma alternativa é implementar os algoritmos de aprendizado por reforço (Reinforcement Learning - *RL*) diretamente em uma linguagem de maior desempenho, como *C/C++*. Isso elimina a necessidade de usar *Python* para o tratamento dos dados e reduz o tempo de execução dos algoritmos. Entretanto, o uso direto de *C/C++* gera dificuldades, tanto em razão da complexidade da implementação, quanto na usabilidade do resultado final, sendo destacado que a configuração das dependências e sua instalação, acabam sendo mais complexas em *C/C++* do que em *Python*. Além disso, o fato de gerenciar a memória conscientemente, acaba aumentando a complexidade de uso das bibliotecas. Desse modo, o surgimento de *bugs* e outros problemas durante o desenvolvimento são potencializados, ocasionando a redução de sua adoção pela comunidade da área já habituada com *garbage collection*, as facilidades de instalação e também de configuração de dependências.

Contudo, apesar dessas dificuldades, ainda há outras alternativas para abordá-las, como é o caso da linguagem *Rust*. Linguagem essa que já demonstrou alta performance em diversos cenários (BUGDEN; ALAHMAR, 2022; PEREIRA et al., 2017), alcançando níveis comparáveis aos programas desenvolvidos em *C/C++*. Além disso, o custo dos *bindings* entre *Rust* e *C/C++* é praticamente inexistente, permitindo assim o uso de bibliotecas como o *TensorFlow* ou o *PyTorch* sem aumentos significativos no tempo de execução. À vista disso, uma outra característica muito importante do *Rust* é seu foco na segurança em gerenciar memória, eliminando a necessidade de um *garbage collector*, rea-

lizando automaticamente a alocação e liberação de memória, sendo quase tudo verificado em tempo de compilação pelo *borrow checker*, reduzindo também a presença de *bugs* e outros problemas na *codebase*.

1.3 Objetivo geral e Objetivos específicos

Dadas essas capacidades que a linguagem *Rust* proporciona, o objetivo geral do trabalho é verificar empiricamente se uma implementação em *Rust* do algoritmo *Deep Q-Network* (*DQN*) é mais rápida do que uma implementação em *Python*.

Para isso, será realizada a comparação do algoritmo *DQN* implementado em *Rust* com a implementação em *Python* fornecida pela biblioteca *StableBaselines3* (RAFFIN et al., 2021), visando analisar se a implementação em *Rust* fornece um menor tempo de treinamento. Para tanto, os objetivos específicos são os seguintes:

1. Implementar em *Rust* o algoritmo *DQN* e também seus *bindings* para *Python*, visando facilitar sua integração com outras bibliotecas em *Python*;
2. Avaliar o desempenho da implementação do *DQN* em *Rust* em comparação com a implementação em *Python*, utilizando o ambiente *Cart-Pole* como cenário de testes;
3. Comparar os resultados de desempenho entre as implementações em *Rust* e *Python*, focando na velocidade e eficiência de execução para determinar qual abordagem oferece maior desempenho.

1.4 Organização do trabalho

O trabalho se organiza da seguinte forma, na primeira seção do capítulo 2 apresenta-se os conceitos básicos do aprendizado por reforço e de redes neurais, assim como de que forma esses dois conceitos são unidos para o aprendizado por reforço profundo.

Na seção seguinte, são apresentadas noções importantes sobre quais os fatores que interferem no desempenho de um programa: a linguagem em que ele é implementado, as otimizações que tal linguagem fornece, a forma como a memória dinâmica é gerenciada e, por fim, o impacto dos *bindings* entre linguagens.

No capítulo 3, são destacados alguns trabalhos relacionados ao tema dessa proposta, os quais são bibliotecas que implementam algoritmos de aprendizado por reforço em lin-

guagens que fornecem melhor desempenho ou que auxiliam, de alguma outra forma, a aumentar o desempenho dos algoritmos de aprendizado por reforço, por exemplo, com ambientes mais rápidos ou facilitando a distribuição da computação. Além dessas, ainda há as bibliotecas que são *baselines* para a comunidade, servindo como ponto de referência para comparações, pois são as mais difundidas nesse meio. Ao fim do capítulo, são resumidas as limitações e vantagens de cada trabalho relacionado.

Em seguida, no capítulo 4, é detalhada a metodologia, explicando o algoritmo *Deep Q-Network* e como ele foi implementado em *Rust*. Neste capítulo também são justificadas as escolhas das métricas utilizadas e do ambiente selecionado para o experimento. Por fim, o experimento em si é detalhado, indicando o hardware e software utilizados e como os resultados das métricas serão avaliados posteriormente.

Posteriormente, no capítulo 5, são exibidos e discutidos os resultados obtidos, juntamente dos testes estatísticos que foram realizados.

Por fim, no capítulo 6, as considerações finais são colocadas, resumindo todos os pontos importantes do trabalho, suas limitações e possíveis continuações do mesmo.

2 Fundamentação Teórica

Nesse capítulo serão apresentados os conceitos fundamentais para a compreensão do presente trabalho. Abordando as noções básicas do aprendizado por reforço, suas formalizações, noções de redes neurais e do aprendizado por reforço profundo. Assim como alguns conceitos de desempenho de software relevantes para o trabalho proposto, sendo eles: as diferenças entre linguagens de programação no que tange otimização de código, as diferentes abordagens para gerência de memória dinâmica e seus impactos e, por fim, o que são e quais são os custos dos *bindings* entre linguagens. Concomitante a tais conceitos serão apresentados os desafios e *trade-offs* de cada um deles, além de como se relacionam com a mudança de implementação dos algoritmos de *Python* para *Rust*, em particular do algoritmo *DQN*.

2.1 Aprendizado por reforço profundo

Segundo o trabalho desenvolvido pela *DeepMind*, responsável por cunhar o termo *deep reinforcement learning*, o aprendizado por reforço profundo surge da união das técnicas de aprendizado por reforço com o uso de redes neurais profundas (*deep learning* em inglês) (MNIH et al., 2013), se fazendo necessário introduzir esses dois conceitos para entender como eles funcionam e, em seguida, o motivo de serem unidos. Isso permitirá o entendimento dos desafios que trazidos por tal união, principalmente em relação aos custos computacionais de tais métodos. Desse modo, ficará clara a necessidade de formas de reduzir o tempo e custo da execução de modelos de aprendizado por reforço profundo, sendo esse o objetivo almejado com a implementação do *DQN* em *Rust*.

2.1.1 Noções básicas de aprendizado por reforço

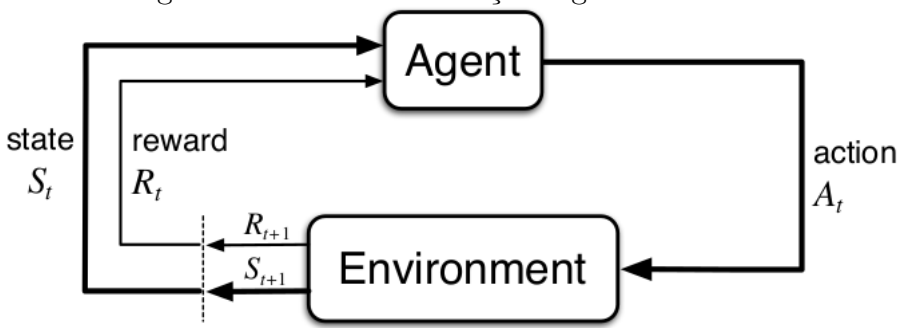
O aprendizado por reforço é uma subárea do aprendizado de máquina que, por sua vez, é um ramo da inteligência artificial, sendo o aprendizado de máquina um campo

dedicado a construir programas que automaticamente ficam melhores com a experiência (MITCHELL, 1997), o aprendizado por reforço busca alcançar tal objetivo ao maximizar um sinal numérico que indica quando uma decisão que leva ao resultado desejado foi executada (SUTTON; BARTO, 2018). Dessa forma, o aprendizado por reforço busca ensinar a um agente, que nada mais é do que uma entidade que possui uma percepção do ambiente, o que fazer em um dado ambiente de forma a maximizar tal sinal numérico de recompensa. O ambiente, nesse caso, é tudo aquilo que é externo ao agente, bem como o seu objeto de interação e a fonte dos estados percebidos pelo agente (SUTTON; BARTO, 2018).

O agente atua no ambiente por meio de um conjunto predefinido de ações, às quais potencialmente modificam o ambiente. Sendo que não é informado ao agente quais ações devem ser realizadas, ou seja, o próprio agente deve descobrir quais ações fornecem as maiores recompensas ao tentá-las (SUTTON; BARTO, 2018). Isto posto, as recompensas são valores numéricos informados ao agente após a execução de uma ação no ambiente, em que esses valores é que irão definir o objetivo a ser resolvido no problema, sendo responsáveis por informar quais eventos são positivos e quais são negativos para o agente (SUTTON; BARTO, 2018).

As relações entre esses componentes são representadas na Figura 1 abaixo:

Figura 1: O ciclo de interações agente-ambiente



Fonte: (SUTTON; BARTO, 2018).

Além do agente, do ambiente, das ações e das recompensas, há outro elemento importante para compreender um sistema de aprendizado por reforço: a política. A política é responsável por definir a forma como o agente age no ambiente em um dado momento. De maneira simplificada, seria um mapeamento entre os estados que o agente percebe do ambiente para com as ações a serem executadas por ele quando está em tais estados (SUTTON; BARTO, 2018). Sendo, dessa forma, a política o que mais caracteriza um agente, pois é com base nela que ele decide quais ações executar no ambiente. O agente aprende ao utilizar os valores obtidos das recompensas para modificar a política, almejando assim

conseguir mais recompensas.

Por exemplo, imagine um ambiente com quatro estados onde o agente pode realizar quatro ações, as políticas para esse ambiente poderiam ser representadas da seguinte forma:

Tabela 1: Exemplo de política tabular com probabilidades para cada uma das quatro ações

Ações / Estado	**W**	**X**	**Y**	**Z**
A	0.6	0.2	0.1	0.1
B	0.2	0.2	0.2	0.4
C	0.2	0.0	0.5	0.3
D	0.0	0.1	0.8	0.1

Na política da tabela 1, para cada um dos quatro estados listados na primeira coluna, A, B, C e D, é informada a probabilidade de cada ação W, X, Y e Z ser a mais indicada para o estado em questão. Essa representação é denominada de política tabular. Porém, uma limitação da representação tabular para as políticas é que cada estado possui de maneira independente suas probabilidades para cada ação. Dessa forma, quanto mais estados possuir um ambiente, maior será a tabela da política e mais difícil será sua atualização. Salientando ainda que em ambientes com estados contínuos, é impraticável utilizar tal representação, pois existiriam infinitos estados a serem representados. Além de que a semelhança entre eles não será considerada para o aprendizado, sendo um problema que aumenta consideravelmente o tempo necessário para o agente aprender.

Compreender tal ponto é relevante para este trabalho, pois é devido a essa limitação da representação tabular que, ao longo do tempo, diversas abordagens foram utilizadas para substituir a representação tabular: Aproximações por funções lineares (JIN et al., 2019); Redes neurais via o clássico Multilayer Perceptron (*MLP*) (TESAURO, 1995); Redes com funções de base radial (*RBF*) (DU et al., 2017); Árvores de decisão (PYEATT, 2003); e Métodos de *ensemble* de modelos.

Tais técnicas resolvem as limitações da representação tabular ao permitir que sejam utilizados valores contínuos como observação para o agente, assim como também podem permitir que o agente generalize qual ação tomar em uma nova observação com base em observações similares vistas anteriormente, algo que não é possível com o uso das politicas tabulares a não ser que a observação percebida pelo agente já seja exatamente a mesma.

Dentre as técnicas mencionadas, se destaca o uso do *MLP* para o problema do Gamão (TESAURO, 1995) que se tornou um marco no uso de redes neurais em aprendizado por

reforço, apenas posteriormente que o uso de redes com múltiplas camadas passou a ser factível com o aumento da disponibilidade de recursos computacionais.

Por fim, apresentados os conceitos básicos do aprendizado por reforço, na seção seguinte, eles serão formalizados para que se possa dar seguimento nos demais conhecimentos essenciais para compreender o problema que está sendo abordado.

2.1.2 Formalização dos conceitos de *RL*

Uma formalização clássica para tomadas de decisão sequenciais é o de Processos de Decisão de Markov (*MDP* em inglês). Em um *MDP*, as ações influenciam tanto as recompensas imediatas quanto os estados subsequentes e, consequentemente, as recompensas futuras, portanto é uma formalização adequada para o aprendizado por reforço (SUTTON; BARTO, 2018). Dessa forma, um *MDP* é uma 4-tupla (S, A, P, R) onde:

- S : é o conjunto de estados do ambiente;
- A : é o conjunto de todas as ações do agente e, em especial, A_s é o conjunto de ações permitidas no estado s;
- P : é o conjunto de probabilidades tais que, a ação a no estado s_t leve para o estado s'_t no momento t seguinte, com os elementos de P sendo denotados por $P_a(s_t, s'_t)$;
- R : é o conjunto de recompensas recebidas após realizar a ação a no estado s_t levando para o estado s'_t no momento t seguinte, com os elementos de R sendo denotados por $R_a(s_t, s'_t)$.

Com a formalização de *MDP*, cobre-se as noções do ambiente (S), das ações (A), das probabilidades de transições do ambiente (P), que geralmente não são utilizadas, pois são comumente desconhecidas e, por fim, das recompensas (R). Logo, resta apenas formalizar a política, que é o principal componente que constitui um agente.

Desse modo, a política π é formalizada como uma distribuição de probabilidade de ações sobre estados. Isto é, a probabilidade de cada ação ser executada dado um estado em particular, de forma que maximize a expectativa de recompensa acumulada.

$$\pi(s, a) \doteq Pr(A_t = a | S_t = s) \tag{2.1}$$

De posse dessa formalização, vale a pena destacar que não existe restrição para a modelagem da política como uma tabela, desde que a política seja uma distribuição de

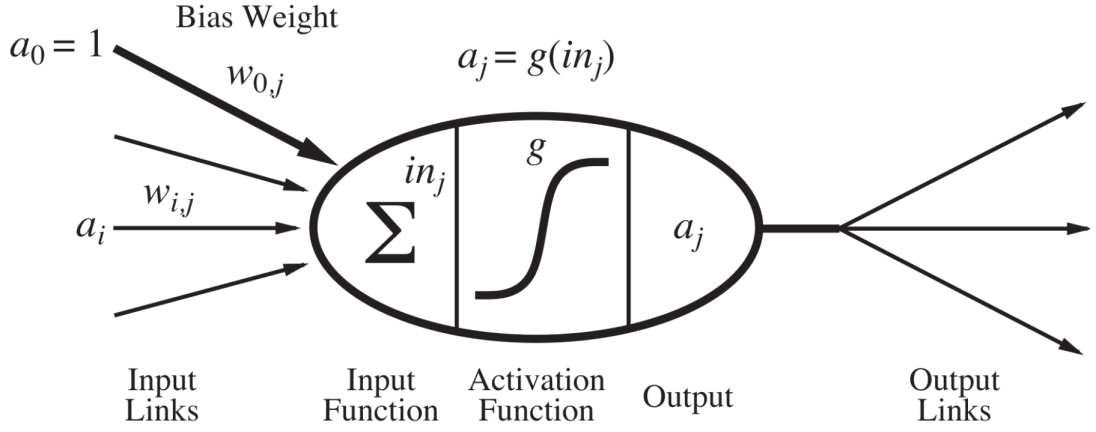
probabilidades de ações sobre estados, com outras representações também se mostrando válidas, assim como mencionado anteriormente com as diversas abordagens que surgiram ao longo dos anos.

2.1.3 Redes neurais artificiais

As redes neurais artificiais são aproximadores de funções compostas por unidades, muitas vezes chamados de neurônios, que são conectadas entre si e cada uma dessas conexões propaga a ativação de uma unidade para a seguinte, ponderando a ativação por um peso. Nesse sentido, para cada unidade é realizada a soma ponderada de todas as conexões de entrada por seus pesos e também de um viés (*bias*), que é um valor extra associado e específico à própria unidade, sendo, enfim, aplicada uma função de ativação no resultado dessa soma (RUSSELL; NORVIG, 2020).

A Figura 2 representa o funcionamento de um neurônio, indicando a soma de seus valores de entrada a_i após serem multiplicados pelos pesos W_{ij} e somados juntamente com o valor de viés W_{0j}, o resultado desse somatório é transformado pela função de ativação g que então é propagado para os demais neurônios conectados a ele na camada seguinte.

Figura 2: Exemplo de modelo matemático de um unidade (neurônios)

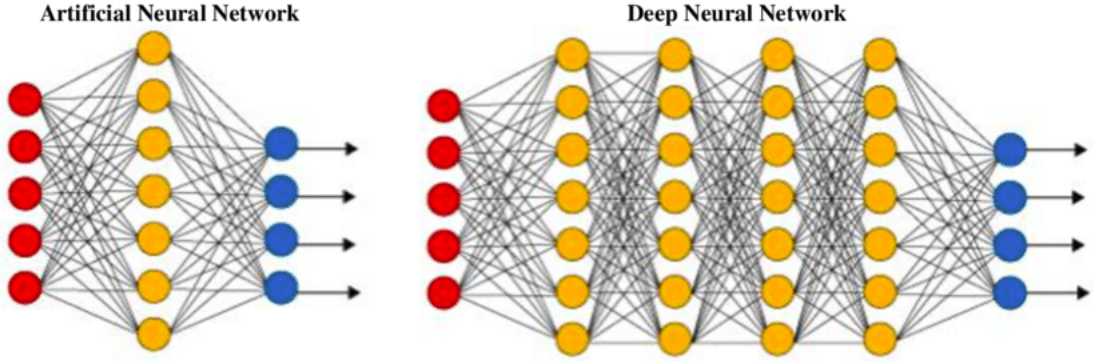


Fonte: (RUSSELL; NORVIG, 2020).

Partindo então dessa ideia base, foram criadas várias configurações para a arquitetura das redes. Em especial, temos a configuração *feed-forward network*, nomeada assim devido as conexões que ocorrem em apenas uma direção. Tal configuração normalmente é organizada em camadas, de forma que as unidades de uma camada enviam seus sinais de ativação para a camada seguinte e, com o aumento do número de camadas, surge o que denomina-se como *deep learning* (GOODFELLOW; BENGIO; COURVILLE, 2016).

Na Figura 3 temos uma representação que ilustra de forma didática a diferença entre redes neurais *shallow* e redes neurais *deep*. Porém é importante destacar que não há consenso no que caracteriza uma rede como profunda, qual seria o *threshold* que delimite uma arquitetura *shallow* de uma *deep* (SCHMIDHUBER, 2015), mesmo que alguns pesquisadores façam uma distinção de que arquiteturas *shallow* consistam de apenas uma camada oculta (*hidden layer*) enquanto que arquiteturas *deep* possuem duas ou mais (GOODFELLOW; BENGIO; COURVILLE, 2016).

Figura 3: Comparação entre uma rede neural "*shallow*" e uma rede neural "*deep*"



Fonte: (MOSTAFA et al., 2020).

De posse dessas noções sobre o que são redes neurais e qual o objetivo delas, que é basicamente permitir a aproximação de funções, pode-se introduzir como ocorre a união entre as redes neurais e o aprendizado por reforço.

As redes neurais são utilizadas como alternativa para a representação das políticas, em que, em vez de usar tabelas são utilizadas redes neurais como estrutura para aproximar a política dos agentes. Porém, as redes neurais simples, de apenas uma camada, não são tão adequadas para representar as políticas para problemas de grande complexidade, podendo causar divergências durante o aprendizado ou impedir que uma solução seja encontrada (TSITSIKLIS; ROY, 1997).

2.1.4 Técnicas para o Aprendizado por Reforço Profundo

O aprendizado por reforço é conhecido por ser instável ou até mesmo divergir quando uma rede neural é usada para representar a política (TSITSIKLIS; ROY, 1997). Essa instabilidade tem várias causas, as correlações presentes na sequência de observações, o fato de que pequenas atualizações na rede podem alterar significativamente a política modificando a distribuição de dados, e as correlações entre os valores utilizados no aprendizado.

Por causa dessa instabilidade diversas técnicas surgiram ao decorrer do tempo para adereçar tal instabilidade, e nesta seção serão apresentadas duas dessas técnicas. Além de discutir qual o impacto que elas tiveram no algoritmo *Deep Q Network* avaliado no presente estudo e que será apresentado logo mais.

A primeira técnica é o *replay buffer*, conhecido também por outros nomes, como *experience replay*, *replay memory* ou *memory buffer*. Essa técnica consiste no armazenamento das experiências passadas do agente, e randomiza os dados, removendo assim as correlações na sequência de observação e suavizando mudanças na distribuição de dados (MNIH et al., 2015).

A segunda técnica é a *target network* que combina duas redes neurais trabalhando em conjunto para atualização da política. Desse modo ocorre uma atualização iterativa que ajusta os valores de ação (*evaluation network*) em direção aos valores-alvo (*target network*), de modo que a *target network* tem seus pesos atualizados periodicamente, reduzindo assim as correlações com o alvo (MNIH et al., 2015).

Foi com a utilização de redes profundas, juntamente de técnicas como essas duas que a representação das políticas por meio de redes neurais se tornou poderosa o suficiente para abordar problemas mais desafiadores (DONG; DING; ZHANG, 2020). Sendo possível visualizar melhor o impacto obtido com o uso dessas técnicas na Tabela 2, onde as recompensas obtidas em cada ambiente com e sem o uso do *replay buffer* e da *target network* são apresentadas quando utilizadas em conjunto com o algoritmo *DQN*.

Tabela 2: Recompensas em 5 jogos do *Atari* com e sem o *replay buffer* e de *target network*

Jogo (Ambiente)	**Ambos**	**Apenas replay buffer**	**Apenas target network**	**Nenhum**
Breakout	316.8	240.7	10.2	3.2
Enduro	1006.3	831.4	141.9	29.1
River Raid	7446.4	4102.8	2867.7	1453.0
Seaquest	2894.4	822.6	1003.0	275.8
Space Invaders	1088.9	826.3	393.2	302.0

Fonte: (MNIH et al., 2015).

Assim, é perceptível que a utilização das duas técnicas proporciona a capacidade de resolução de problemas mais complexos como os jogos do Atari, pois ao não utilizar nenhuma das duas técnicas não se obtinha recompensas expressivas.

Essa alteração na representação da política permite uma maior generalização dos

estados do ambiente para a política. Não sendo necessário que exista uma entrada na tabela para cada estado diferente, permitindo que estados similares sejam agrupados e que a experiência do agente, em diferentes estados, seja usada na atualização da política.

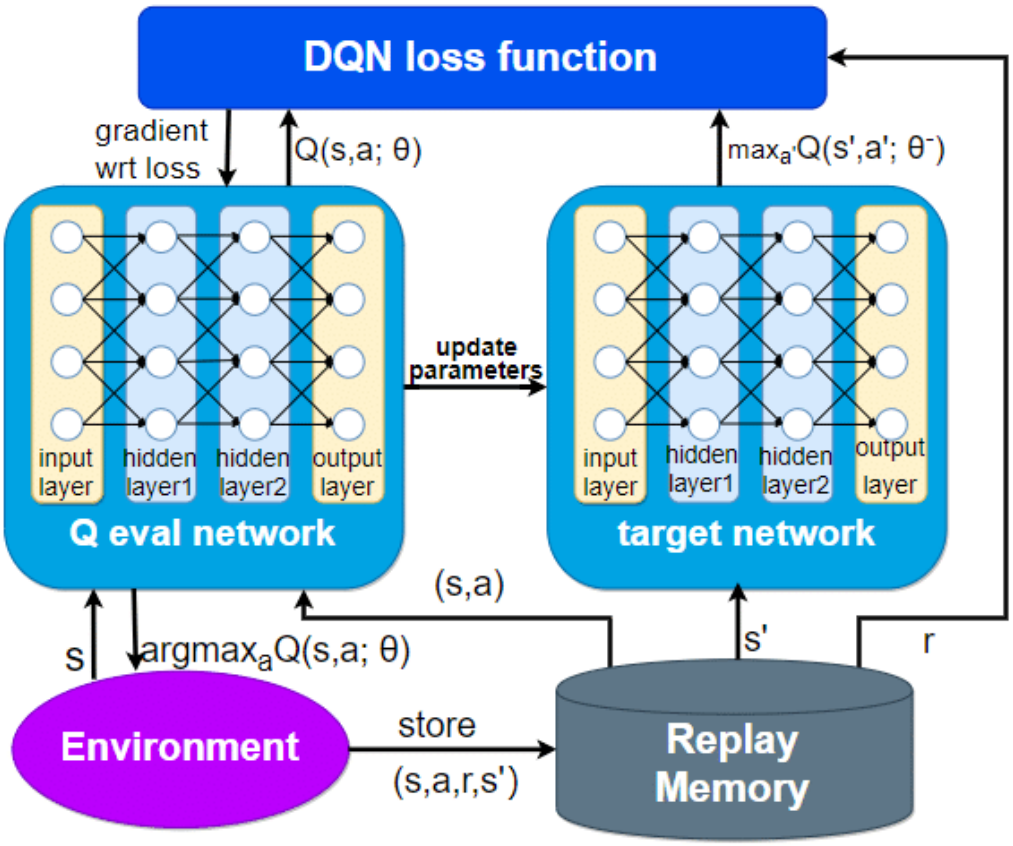
2.1.5 O Algoritmo Deep Q Network

O algoritmo *Deep Q-Network* foi apresentado inicialmente em 2013 (MNIH et al., 2013) e melhorado em 2015 (MNIH et al., 2015) sendo aplicado em diversos jogos de Atari, superando o desempenho de todos os algoritmos anteriores e atingindo um nível comparável ao de um jogador profissional em um conjunto de 49 jogos, usando a mesma arquitetura de rede e hiper-parâmetros.

O *DQN* modifica o algoritmo *Q-learning* (SUTTON; BARTO, 2018) ao utilizar uma rede neural profunda para aproximar a política. Ele mantém a mesma estratégia para atualização da política, isto é, seu valor alvo é dado por: $r+\gamma max_{a'}Q(s',a')$, onde $max_{a'}Q(s',a')$ seria o mesmo de ter executado a melhor ação conhecida para o estado seguinte s'.

A Figura 4 apresenta um diagrama que sintetiza o algoritmo *DQN* clássico:

Figura 4: Diagrama do funcionamento do *DQN*



Fonte: (ZHOU et al., 2019)

A *Q eval network* é responsável por interagir com o ambiente e também é ela que tem seus pesos atualizados a cada rodada de otimização, o *replay memory* armazena as experiencias do agente como tuplas armazenando o estado s, a ação executada a, a recompensa r e o estado seguinte s'. Sendo que após determinada quantidade de experiencias serem armazenadas, é selecionada uma amostra aleatória desse *buffer* e usada para atualizar os pesos da *Q eval network*, porém a resposta "correta" com a qual a saída da *Q eval network* é comparada é a saída da *target network* que não foi atualizada tão frequentemente, reduzindo assim a instabilidade do alvo. Enquanto que a *target network* tem seus pesos atualizados também após determinada quantidade de interações com o ambiente, de forma que os pesos da *Q eval network* são copiados para a *target network*.

Com base no que foi exposto fica claro que existe todo um controle especifico dos métodos de aprendizado por reforço além do treinamento das redes neurais. De forma que toda essa computação tem um custo, a seção seguinte explora os custos associados a utilização do aprendizado por reforço profundo, indicando também quais as possíveis abordagens para reduzir alguns deles.

2.1.6 Desafios e Custos do Aprendizado por Reforço Profundo

Devido a maior capacidade de generalização proporcionada pelas redes neurais profundas, outro desafio surge, pois ao permitir que problemas maiores e mais complexos sejam abordados pelo aprendizado por reforço, ocorre um aumento no custo computacional com a execução das redes neurais artificiais. Nesse sentido, com a possibilidade de abordar problemas mais complexos, quanto mais complexos os ambientes, mais custosa se torna a interação com seus ambientes, necessitando também de um maior tempo de treinamento para lidar com a complexidade do ambiente.

Por exemplo, o *AlphaGo Zero* (SILVER et al., 2017), um agente de aprendizado por reforço profundo criado para jogar o jogo de Go, foi treinado por cerca de 40 dias utilizando 64 *workers* com *GPU* e 19 servidores de parâmetros só com *CPU*, além de 4 *TPUs* para realizar as inferências da rede. Tal treinamento exigiu um investimento de aproximadamente 25 milhões de dólares em *hardware AlphaGo Zero* (GIBNEY, 2017), enquanto que em um *hardware* mais modesto o treinamento teria sido muito mais longo. Dessa forma, reduzir o custo computacional também traz outros benefícios, como uma redução nos custos de *hardware* necessários para a resolução de novos problemas. Portanto, a redução dos custos computacionais na execução de métodos de aprendizado por reforço profundo é um objetivo altamente desejado.

Isso posto, há diversas maneiras de reduzir os custos computacionais, destacando tanto a criação de novos algoritmos e estratégias para reduzir a interação com o ambiente ou acelerar a obtenção de uma política melhor, quanto a própria proposta deste trabalho, que visa otimizar o uso dos recursos computacionais por meio de uma implementação mais eficiente. Portanto, se faz necessário entender alguns dos fatores que afetam o desempenho dos algoritmos atualmente usados no aprendizado por reforço profundo, bem como quais são as alternativas existentes para abordar esses pontos.

2.2 Aspectos de desempenho relacionados ao DRL

Nesta seção, serão abordadas algumas das características envolvidas no desempenho de um software, em especial as mais relevantes para que seja compreendido como o presente trabalho busca melhorar o desempenho dos métodos de aprendizado por reforço, por meio da linguagem *Rust*.

2.2.1 Tipos de linguagens de programação e seus respectivos desempenhos

Inicialmente, deve-se compreender que, para um programa ser executado, ele deve ser primeiro traduzido para código de máquina. Nesse sentido, existem três diferentes abordagens para essa tradução, que são a compilação e a interpretação, além da hibridização entre essas duas abordagens (AHO et al., 2006).

Em linguagens compiladas, o código-fonte é traduzido pelo compilador para o código de máquina de uma plataforma em particular, com instruções especificas para o processador onde o programa será executado. Dessa forma, os programas compilados são significativamente mais rápidos e eficientes em sua execução (AHO et al., 2006). Uma outra característica é que por serem linguagens estaticamente tipadas, os erros de tipo são identificados no estágio da compilação, evitando problemas no momento da execução (ERNEST; MENSAH; GILBERT, 2017). Para tanto, as linguagens *C*, *C++* e *Rust* são linguagens compiladas, ou seja, possuem tais características e, para além disso, as bibliotecas de redes neurais como o *TensorFlow* e *PyTorch* foram desenvolvidas em *C++* justamente por abordarem o processo de otimização utilizado no treinamento de redes neurais, fator crítico para a performance de programas que dependem dessas redes.

Já em relação às linguagens interpretadas, a tradução para o código de máquina ocorre concomitantemente com a execução do programa. Isto é, conforme cada instrução

no código-fonte é alcançada, ela é interpretada e imediatamente executada. Dessa forma, programas escritos nessas linguagens são facilmente portáveis entre diferentes plataformas, bastando ter o interpretador instalado e o acesso ao código-fonte.

Entretanto, tal flexibilidade vêm com um custo, pois não é possível realizar tantas otimizações, dado que cada instrução do código-fonte tem que ser interpretada toda vez que é alcançada. Um outro custo é que os tipos em linguagens interpretadas são verificados durante a própria execução, aumentando ainda mais o tempo de execução (ERNEST; MENSAH; GILBERT, 2017). Com isso, é importante destacar que a linguagem *Python* é interpretada e, atualmente, a grande maioria das bibliotecas de aprendizado por reforço têm seus algoritmos escritos nessa linguagem. Por isso, utilizar linguagens compiladas como *Rust* é mais vantajoso nesse tipo de situação, dado que se consegue gerar código muito mais otimizado do que *Python*.

Por fim, para as linguagens hibridas, é feito um meio-termo entre as duas abordagens, o código-fonte é compilado em uma linguagem intermediária e essa linguagem intermediária é interpretada. Tal abordagem tenta alcançar um equilíbrio entre otimizações e flexibilidade, em que, para o passo de compilação as oportunidades de otimizações são identificadas, gerando o código intermediário específico para maior desempenho, enquanto que com a etapa de interpretação a flexibilidade de execução em múltiplas plataformas é garantida (ERNEST; MENSAH; GILBERT, 2017). Desse modo, linguagens que seguem essa abordagem também podem fazer uso dos chamados *just-in-time compilers*, compilando partes críticas do programa durante sua execução, já que, por meio da análise do programa em execução, conseguem identificar novas oportunidades de otimização (AHO et al., 2006). Todavia, utilizar linguagens híbridas para tentar reduzir o custo computacional no aprendizado por reforço profundo pode não ser interessante, pois, em geral, elas não alcançam o mesmo desempenho de linguagens compiladas como o *Rust*, *C* ou *C++*, sobretudo porque possuem o passo de interpretação do código intermediário.

Para comprovar tais afirmações, utiliza-se como base um estudo comparativo entre diversas linguagens realizado em um conjunto variado de problemas, conforme os resultados apresentados na Figura 5 que contém uma tabela com os dados normalizados para o consumo de energia, tempo e memória (PEREIRA et al., 2017).

Figura 5: Comparação entre linguagens por consumo de energia, tempo e memória

	Energy
(c) C	1.00
(c) Rust	1.03
(c) C++	1.34
(c) Ada	1.70
(v) Java	1.98
(c) Pascal	2.14
(c) Chapel	2.18
(v) Lisp	2.27
(c) Ocaml	2.40
(c) Fortran	2.52
(c) Swift	2.79
(c) Haskell	3.10
(v) C#	3.14
(c) Go	3.23
(i) Dart	3.83
(v) F#	4.13
(i) JavaScript	4.45
(v) Racket	7.91
(i) TypeScript	21.50
(i) Hack	24.02
(i) PHP	29.30
(v) Erlang	42.23
(i) Lua	45.98
(i) Jruby	46.54
(i) Ruby	69.91
(i) Python	75.88
(i) Perl	79.58

	Time
(c) C	1.00
(c) Rust	1.04
(c) C++	1.56
(c) Ada	1.85
(v) Java	1.89
(c) Chapel	2.14
(c) Go	2.83
(c) Pascal	3.02
(c) Ocaml	3.09
(v) C#	3.14
(v) Lisp	3.40
(c) Haskell	3.55
(c) Swift	4.20
(c) Fortran	4.20
(v) F#	6.30
(i) JavaScript	6.52
(i) Dart	6.67
(v) Racket	11.27
(i) Hack	26.99
(i) PHP	27.64
(v) Erlang	36.71
(i) Jruby	43.44
(i) TypeScript	46.20
(i) Ruby	59.34
(i) Perl	65.79
(i) Python	71.90
(i) Lua	82.91

	Mb
(c) Pascal	1.00
(c) Go	1.05
(c) C	1.17
(c) Fortran	1.24
(c) C++	1.34
(c) Ada	1.47
(c) Rust	1.54
(v) Lisp	1.92
(c) Haskell	2.45
(i) PHP	2.57
(c) Swift	2.71
(i) Python	2.80
(c) Ocaml	2.82
(v) C#	2.85
(i) Hack	3.34
(v) Racket	3.52
(i) Ruby	3.97
(c) Chapel	4.00
(v) F#	4.25
(i) JavaScript	4.59
(i) TypeScript	4.69
(v) Java	6.01
(i) Perl	6.62
(i) Lua	6.72
(v) Erlang	7.20
(i) Dart	8.64
(i) Jruby	19.84

Fonte: (PEREIRA et al., 2017).

Na Figura 5 as linguagens compiladas são identificadas por um *(c)*, as interpretadas por um *(i)* e as hibridas por um *(v)*, de uma maquina virtual. Descrita a tabela, é possível identificar que o topo das três métricas é dominado por linguagens compiladas, enquanto que as linguagens interpretadas ocupam principalmente as últimas posições nesses três quesitos e as linguagens hibridas habitam as posições intermediárias.

Dessa forma, a linguagem *Python* está nas duas últimas posições, tanto no consumo de energia quanto no gasto de tempo, além de ocupar a 12ª posição no consumo de memória. Por outro lado, a linguagem *Rust* encontra-se em segundo lugar tanto no consumo de energia, quanto no consumo de tempo, ficando, inclusive, acima de *C++*, perdendo apenas para a linguagem *C* e, estando na 7ª posição no consumo de memória, apresentando melhor performance do que *Python* (PEREIRA et al., 2017).

2.2.2 Alterações no desempenho a partir da gerência de memória

A gerência de memória dinâmica de um programa consiste principalmente em dois aspectos, a alocação de um novo bloco de memória e a desalocação de memória que não será mais usada (KNUTH, 1997). A forma como as linguagens lidam com esses aspectos afetam o desempenho dos programas, pois a alocação dinâmica de memória é um dos recursos mais utilizados e, devido ao custo que é dado ao acessar a memória por questões relacionadas ao *hardware* dos sistemas computacionais, garantir o bom funcionamento e segurança dessa gerência é fundamental (JONES; HOSKING; MOSS, 2023).

À vista disso, a gerência da memória, mesmo aparentando ser simples como algo que apenas aloca memória quando necessário e desaloca quando não estiver mais usando, na realidade não é, principalmente em programas complexos, tanto que algumas linguagens se utilizam de um *garbage collector*, uma rotina que, em determinados momentos da execução de um programa, libera os blocos de memória que não são mais utilizados pelo programa, como forma de evitar que os programadores tenham que lidar diretamente com tal gerência (JONES; HOSKING; MOSS, 2023). Porém, o uso de tais rotinas afeta o desempenho do programa, pois se faz necessário pausar sua execução para realizar a rotina de limpeza da memória. Nesse sentido, é importante ressaltar que a linguagem *Python* faz uso de *garbage collector*, sendo esse outro fator para a redução de sua performance.

Desse modo, linguagens com *garbage collector* não se apresentam adequadas para programas que necessitam de alta performance, sendo comumente recomendadas linguagens como *C*, *C++* e *Fortran* para abordar situações de alta performance. Entretanto essas linguagens têm um sistema de gerência de memória manual, sendo necessário alocar e desalocar a memória manualmente, mesmo que, com o uso de ponteiros inteligentes (*smart pointers*), a complexidade da gerência seja reduzida e assim as possibilidades de erros de gerência diminua, seu uso ainda não garante a segurança na gerência da memória, pois erros como acesso a ponteiros nulos ainda são um risco (MEYERS, 2014).

Diante disso, a linguagem *Rust* surge como uma opção apropriada, pois fornece segurança na gerência da memória mesmo sem uso de um *garbage collector*. Para isso, a linguagem modela o conceito de posse da memória de maneira explicita no compilador, de modo que a memória é liberada automaticamente quando for apropriado, por exemplo no fim de um escopo ou ao fim da vida (*lifetime*) do objeto. Outro fator que contribui para tal segurança é que não existe o *NULL* em *Rust*, sendo o sistema de tipos, especificamente a enumeração *Option*, responsável por indicar a presença ou não de um valor, evitando, assim, a necessidade do *NULL* (KLABNIK; NICHOLS, 2018).

2.2.3 *Bindings* entre linguagens e seus custos para o DRL

O *binding* entre linguagens é a forma pela qual um recurso, como uma função ou método, definido de forma independente da linguagem, é implementado em uma linguagem de programação específica (MEEK, 1993).

Muitos padrões computacionais são especificados como Interfaces de Programas de Aplicações (*Application Program Interfaces* ou *APIs*). Essas *APIs* são mais comumente expressas como um conjunto de operações, de definições de dados associadas ou como uma semântica das operações em algum sistema subjacente (EMERY, 1996). Dessa forma, as *APIs* fornecem a definição de recursos ou de conceitos, de uma maneira que independe de qualquer linguagem, enquanto que os *bindings* seriam a implementação de uma *API* em termos de uma linguagem de programação especifica (EMERY, 1996).

Diante disso, por meio dos *bindings*, é possível escrever um programa em *Python* e utilizar um recurso implementado em *C* dentro desse programa, desde que exista um *binding* de tal recurso, disponível para *Python*. É desse modo que as bibliotecas de redes neurais implementadas em *C++* conseguem ser utilizadas em *Python*. Porém, os *bindings* entre *Python* e *C* possuem custos, tanto da própria invocação das funções quanto do processo de *marshalling* (TURNER-TRAURING, 2023), termo que indica o processo de transformar a representação de um dado na memória em um formato neutro (TANENBAUM; STEEN, 2023). Salientando que esse processo é importante, pois as formas como *Python* e *C* representam seus dados são diferentes, necessitando da conversão para que a comunicação entre as duas linguagens ocorra (TURNER-TRAURING, 2023).

Outro aspecto a ser levado em consideração é a gerência de memória, noção já apresentada previamente neste capítulo, pois *Python* utiliza um *garbage collector* enquanto que a linguagem *C* não. Assim, deve-se levar em consideração a origem de cada dado para evitar problemas de vazamento ou corrupção de memória, devido a uma falta ou um erro de desalocação de memória (ANDERSON, 2020).

Já nos *bindings* entre *Rust* e *C*, a situação é diferente, pois chamadas de funções implementadas em *C*, a partir de um programa desenvolvido em *Rust*, têm custo zero (CRICHTON, 2015). Claro, o mesmo não ocorre em todos os *bindings* para *C++*, pois alguns dos conceitos e funcionalidades entre as duas linguagens diferem consideravelmente. Nesses casos, é necessário realizar conversões dos dados ou até mesmo outros artifícios para certas funcionalidades, como por exemplo, o conceito de herança presente em *C++* e inexistente em *Rust* (HUNGER, 2022).

Por fim, ao utilizar *Python*, o ideal é evitar realizar muitas chamadas aos *bindings* de uma biblioteca em outra linguagem. Caso contrário, os custos de invocação das funções e do *marshalling* podem acabar por comprometer os potenciais ganhos de desempenho do uso da biblioteca externa. Dessa forma, concentrar a computação dentro das bibliotecas externas e representar os dados de forma a serem compatíveis com a linguagem externa são formas de reduzir os impactos dos *bindings* (TURNER-TRAURING, 2023). Por outro lado, os *bindings* de *Rust* para outras linguagens são bem menos custosos, principalmente para a linguagem *C*, com o custo zero. Por fim, é importante mencionar também que é possível criar *bindings* entre *Python* e *Rust*, porém com um custo similar aos *bindings* entre *Python* e *C++*.

3 Trabalhos Relacionados

Neste capítulo são apresentados trabalhos relevantes para a execução de experimentos de aprendizado por reforço, sejam trabalhos que tiveram ampla adoção pela comunidade e portanto pertinentes para o presente trabalho, por servirem de *baseline* para possíveis comparações. Além disso, também se inserem trabalhos que se enquadram em uma das duas diferentes abordagens para otimização dos experimentos, acelerar a simulação dos ambientes ou acelerar os algoritmos em si.

3.1 Bibliotecas com ambientes para *RL*

Nesta seção são apresentados trabalhos cujo objetivo é fornecer implementações de ambientes que implementem problemas para *RL*. Em especial trabalhos que busquem reduzir o tempo de execução dos métodos de *RL* por meio de ambientes mais eficientes, pois ao reduzir o custo da interação com o ambiente se obtém um menor tempo de execução.

3.1.1 *Gymnasium*

A biblioteca *Gymnasium* é um *fork* mantido da biblioteca *Gym* (BROCKMAN et al., 2016) da *OpenAI*, trazendo muitas melhorias e atualizações de *API* para permitir seu uso contínuo para pesquisa *RL* de código aberto (TOWERS et al., 2024). O *Gymnasium* mantém seu foco inteiramente nos ambiente de pesquisa em *RL*, abstraindo o aspecto do design e implementação dos agentes. A única restrição ao agente é que ele deve fornecer uma ação válida conforme especificado pelo espaço de ações do ambiente (TOWERS et al., 2024).

Além disso, a interface do ambiente do *Gymnasium* é agnóstica à implementação interna da lógica do ambiente, permitindo o uso de programas externos. O único requisito é que o ambiente seja subclasse de *gym.Env* implementando as funções *reset* e *step*

(TOWERS et al., 2024).

3.1.2 *Ray RLlib*

A *RLlib* é uma biblioteca que fornece primitivas de software escaláveis para aprendizado por reforço. Essas primitivas permitem que algoritmos sejam implementados com alto desempenho, escalabilidade e reutilização substancial de código (LIANG et al., 2018). Porém, esta abordagem tem um custo elevado devido ao uso de uma configuração distribuída. Em tal configuração a comunicação envolvida entre os componentes que não são otimizados especificamente para execução do ambiente acaba tendo impacto relevante no tempo de execução.

Os autores da biblioteca conduziram um estudo comparativo que demonstra os ganhos de desempenho da biblioteca, porém o foco foi na escalabilidade da solução para uma grande quantidade de recursos computacionais. Eles chegaram a utilizar até 8192 *CPUs* em alguns experimentos e 512 *CPUs* e 64 *GPUs* em outros, com o objetivo de demonstrar a escalabilidade da solução.

Dessa forma, nota-se que o foco da *RLlib* é a exploração de uma quantidade vasta de recursos computacionais, e não da máxima exploração do recursos disponíveis. A biblioteca disponibiliza além de ambientes algoritmos também, tendo foco na distribuição da computação.

3.1.3 *Isaac Gym*

A biblioteca *Isaac Gym* é uma plataforma de aprendizagem de alto desempenho para treinar políticas em uma ampla variedade de tarefas robóticas inteiramente em *GPU* (MAKOVIYCHUK et al., 2021). Devido à natureza altamente paralela das *GPUs*, vários ambientes podem ser executados simultaneamente. Porém, com essa configuração não é possível lidar com ambientes de jogos virtuais, que são importantes *benchmarks* da comunidade, como os do *Atari*, *Starcraft II* ou *Dota 2*. Isso ocorre pois, as simulações nos ambientes do *Isaac Gym* são voltadas para o mundo físico, enquanto que jogos como *StarCraft II* ou *Dota 2* operam em um mundo mais abstrato e estratégico.

Diferentemente da *RLlib*, a *Isaac Gym* têm como foco a exploração ao máximo das *GPUs*, porém tal foco acaba impedindo que a biblioteca seja utilizada em contextos mais amplos dos problemas do aprendizado por reforço. De toda forma, ela é indicada para ambientes que podem ser simulados totalmente computacionalmente ela é muito indicada.

3.1.4 *envpool*

A biblioteca *envpool* conta com um design selecionado para ambientes paralelos, para qual os autores afirmam que aumenta a velocidade de simulação dos ambiente em diferentes configurações de *hardware*, desde um laptop e uma estação de trabalho modesta até máquinas de última geração (WENG et al., 2022).

A biblioteca é implementada em *C++* e os autores afirmam que os ambientes implementados atualmente em *Python* não podem ser acelerados pela biblioteca (WENG et al., 2022). Fato esse que demonstra outra limitação da utilização de *Python* como linguagem de escolha quando desempenho é uma característica relevante para a solução do problema.

3.2 Bibliotecas com algoritmos para *RL*

Nesta seção são apresentados trabalhos cujo objetivo é fornecer implementações de algoritmos relevantes para *RL*. Em especial trabalhos que busquem reduzir o tempo de execução dos métodos de *RL* por meio de algoritmos inovadores ou otimizados, ao serem implementados em linguagens com maior performance.

3.2.1 *Stable Baselines3*

Stable Baselines3 (*SB3*) é uma biblioteca popular para aprendizado de reforço profundo que oferece uma coleção de implementações de código aberto de alta qualidade de algoritmos construídos sobre o *PyTorch*. Ela fornece uma interface amigável, ampla documentação e exemplos, tornando-a acessível tanto para iniciantes quanto para pesquisadores experientes (RAFFIN et al., 2021).

Stable Baselines3 suporta sete algoritmos de *DRL* diferentes, incluindo o *DQN*. Ela também inclui seis outros algoritmos em seu repositório de algoritmos experimentais, com implementações menos maduras. A biblioteca prioriza estabilidade, eficiência e escalabilidade, e oferece implementações bem testadas de algoritmos garantindo desempenho confiável em vários ambientes e cenários (RAFFIN et al., 2021).

No entanto, nenhum estudo foi conduzido apresentando o tempo de execução dos algoritmos na biblioteca, ou comparando o desempenho dessas implementações com outras bibliotecas e *frameworks* comuns na comunidade de *DRL*.

3.2.2 *smarties*

smarties é uma biblioteca com implementações *C++* de alto desempenho para diversos algoritmos de aprendizado por reforço profundo (NOVATI; KOUMOUTSAKOS, 2019). Ela possui suporte para ambientes com uma *API* similar ao *gym* da *OpenAI*, onde o ambiente possui uma função que, ao receber uma nova ação do agente, avança a simulação no tempo e retorna uma nova observação, junto com a recompensa. Assim como também dá suporte para *APIs* em que a simulação do ambiente domina a estrutura do código da aplicação. Nessa configuração, é o código *RL* que é chamado sempre que novas observações estão disponíveis, exigindo que o agente realize uma ação (NOVATI; KOUMOUTSAKOS, 2019).

Essa biblioteca possui uma interface *CLI* e também *bindings* para *Python*, assim como foi utilizada pelos autores, em quatro estudos, para a resolução de diferentes problemas (NOVATI et al., 2017; VERMA; NOVATI; KOUMOUTSAKOS, 2018; NOVATI; MAHADEVAN; KOUMOUTSAKOS, 2019; NOVATI; KOUMOUTSAKOS, 2019). Mas, infelizmente, não foi realizado nenhum estudo apresentando o ganho de desempenho da biblioteca em relação a uma implementação em *Python* ou que apresente os tempos de execução dos algoritmos.

3.2.3 *CleanRL*

A *CleanRL* é uma biblioteca de aprendizado por reforço profundo baseada em implementações de arquivo único para ajudar pesquisadores a entender todos os detalhes de um algoritmo, prototipar novos recursos, analisar experimentos e dimensionar os experimentos com facilidade (HUANG et al., 2022).

A *CleanRL* não é uma biblioteca modular, cada um dos nove algoritmos implementados e suas variantes são autocontidos em arquivos únicos, entre eles, o algoritmo *DQN* no qual as linhas de código foram reduzidas ao mínimo. Junto com implementações sucintas, a base de código do *CleanRL* é completamente documentada e comparada para garantir que o desempenho esteja no mesmo nível de fontes confiáveis (HUANG et al., 2022).

Os autores não fizeram um estudo comparativo sobre o desempenho das implementações no *CleanRL*, mas eles apresentam alguns tempos de treinamento, mostrando que suas implementações são mais rápidas do que os algoritmos *Stable Baselines3*.

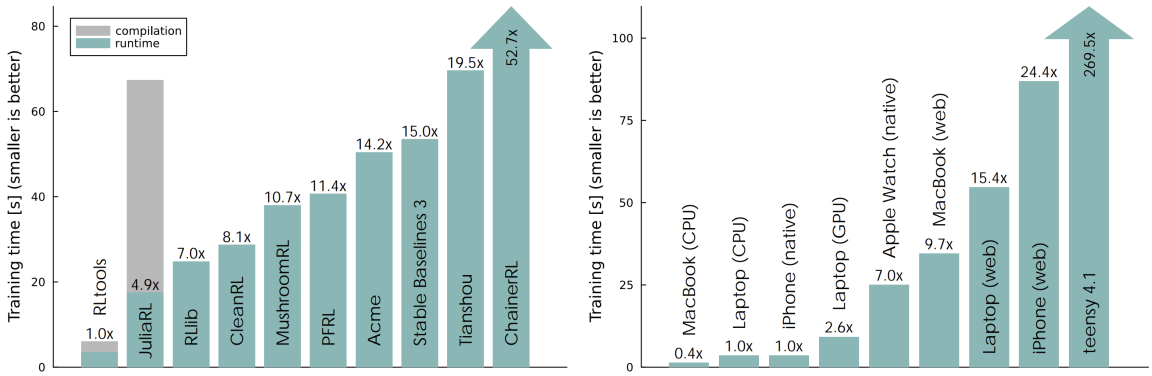
3.2.4 *RLtools*

RLtools se apresenta como uma biblioteca rápida e portátil de aprendizagem por reforço profundo para controle contínuo, também implementada em *C++* (ESCHMANN; ALBANI; LOIANNO, 2023).

Essa biblioteca possui suporte para apenas três algoritmos: *Twin Delayed Deep Deterministic* (*TD3*) , *Proximal Policy Optimization* (*PPO*) e *Soft Actor-Critic* (*SAC*) , implementando seus três próprios ambientes, baseados em problemas clássicos da área: *Pendulum*, *Car* e *Acrobot*. Além dissoe, ela possui uma interface própria para utilização dos ambientes do simulador *MuJoCo* que, segundo os autores, é cerca de 25% mais rápida que a implementação disponível na biblioteca *EnvPool* (ESCHMANN; ALBANI; LOIANNO, 2023). A *RLtools* também implementa sua própria coleção de algoritmos de redes neurais e não possui nenhuma dependência obrigatória de outras bibliotecas, como *PyTorch* ou *TensorFlow* para realização das computações necessárias em tais algoritmos (ESCHMANN; ALBANI; LOIANNO, 2023).

Todavia, diferentemente das bibliotecas anteriores, a *RLtools* apresenta um estudo sobre qual foi o ganho de performance da implementação em *C++*, realizando a comparação com diversas bibliotecas, em diferentes linguagens e também em diferentes dispositivos, como se pode ver na Figura 6 a seguir, onde é apresentado um dos resultados comparativos com as outras bibliotecas.

Figura 6: *Benchmarks* de treinamento no ambiente *Pendulum* comparando outras bibliotecas com a *RLtools*



Fonte: (ESCHMANN; ALBANI; LOIANNO, 2023).

A *RLtools* possui *bindings* para *Python*, mas para obtenção da performance mostrada em suas comparações, se deve implementar os ambientes com a *API* da biblioteca, o que aumenta a complexidade de utilização, pois para extrair o máximo de performance é necessário sacrificar a facilidade de utilização da biblioteca pelo público em geral.

3.3 Panorama Atual e Contribuição do Estudo

Dessa forma, diante do exposto, é importante destacar que essas ultimas quatro bibliotecas, que foram apresentadas, são as principais bibliotecas para o aprendizado por reforço que se assemelham, em algum aspecto, com o presente trabalho, em que cada uma possui seus pontes fortes e fracos. Neste contexto a *Stable Baselines3* é a principal biblioteca usada como *baseline* para comparações de desempenho. Segue abaixo na Tabela 3 os oito trabalhos expostos, sintetizando brevemente suas limitações e vantagens:

Tabela 3: Pontos fortes e pontos fracos dos oito trabalhos expostos

Trabalho	**Limitações**
Gymnasium	Não foca na performance dos ambientes fornecidos
Ray RLlib	Baixa performance para hardwares mais simples
Isaac Gym	Impossibilidade de simular ambientes de jogos virtuais
envpool	Não proporciona paralelização via GPU para os ambientes
Stable Baselines3	Não foca na performance dos algoritmos fornecidos
smarties	Interface de uso é mais complicada do que outras alternativas
CleanRL	Não fornece uma interface unificada para os algoritmos
RLtools	Depende de ambientes próprios para máxima performance
Trabalho	**Vantagens**
Gymnasium	Disponibiliza diversos ambientes com uma API sólida
Ray RLlib	Alta escalabilidade via distribuição em larga escala
Isaac Gym	Paralelização massiva via GPU
envpool	Paralelização genérica com CPU (diversos hardwares)
Stable Baselines3	Disponibiliza diversos algoritmos com implementações robustas
smarties	Dois tipos de API: agent→env & env→agent
CleanRL	Algoritmos de DRL em arquivos únicos e simples
RLtools	Implementação mais eficiente entre os trabalhos encontrados

O presente trabalho busca seguir o padrão da *API* do *Gymnasium* para interação com o ambiente, permitindo assim maior compatibilidade e ampliar seu uso. Para isso, inspira-se nas implementações da *CleanRl* e da *Stable Baselines3*, que são consideradas *baselines* para os diversos algoritmos que são fornecidos pela biblioteca, mas ao mesmo tempo também são considerados aspectos de desempenho assim como a *RLtools*, porém, seguindo o exemplo da *smarties*, fornecendo *bindings* para *Python*.

4 Metodologia

Nesse capítulo é apresentada a implementação do algoritmo *DQN* detalhando as decisões tomadas para sua implementação em *Rust* bem como para a criação dos bindings do que foi implementado para *Python* e como as redes neurais foram treinadas. Em seguida, é apresentada a metodologia experimental adotada, envolvendo a apresentação do ambiente selecionado para a execução dos experimentos, assim como a exposição da configuração utilizada, o que abrange os hiper-parâmetros utilizados em cada algoritmo, assim como seus valores. Em seguida, se expõe a configuração de *hardware* e *software* utilizados. Por fim, as métricas e as justificativas para suas escolhas, bem como a forma de avaliação dos resultados dos experimentos realizados.

4.1 Implementação do algoritmo DQN

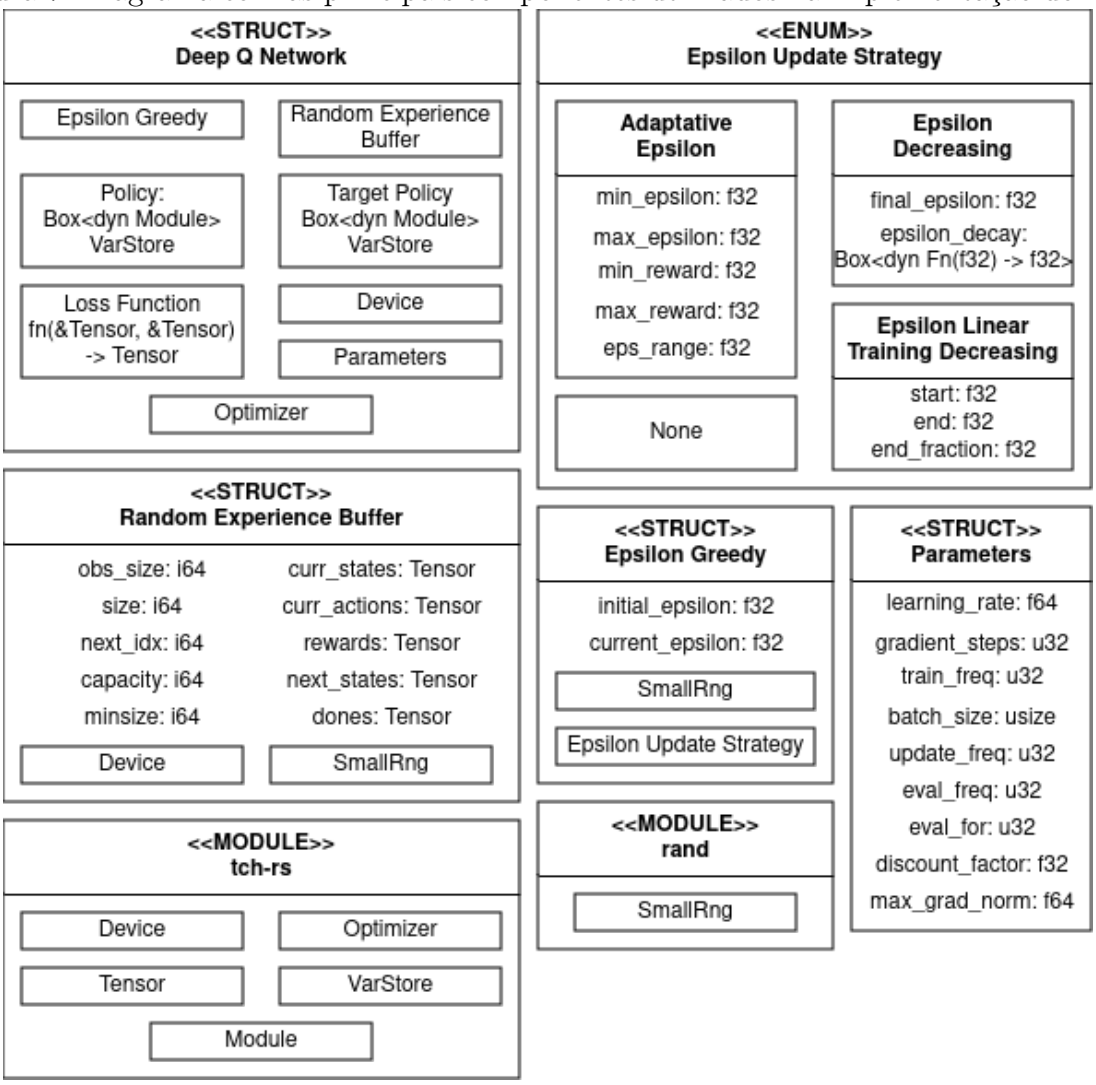
A implementação do algoritmo *DQN* foi separada em cinco componentes:

1. *Deep Q Network*: Componente com a lógica de treinamento e avaliação do algoritmo;
2. *Parameters*: Componente que armazena os hiper-parâmetros do algoritmo *DQN*;
3. *Epsilon Greedy*: Componente responsável pela exploração de novos estados;
4. *Epsilon Update Strategy*: Componente com estratégia de atualização do *epsilon*;
5. *Random Experience Buffer*: Componente que armazena as experiências do agente.

Enquanto que tais componentes foram encapsulados em um *wrapper* que expõe uma interface simplificada, seguindo o padrão da biblioteca *Stable Baselines 3*, onde os bindings para *Python* fora criados, permitindo a utilização da implementação em *Rust* em um script *Python*, possibilitando assim a utilização de todo o ecossistema da comunidade para uso de ambientes e recursos auxiliares de aprendizado por reforço.

Na Figura 7, o diagrama representa os atributos de cada componente e seus tipos, bem como suas dependências.

Figura 7: Diagrama com as principais componentes utilizados na implementação do DQN

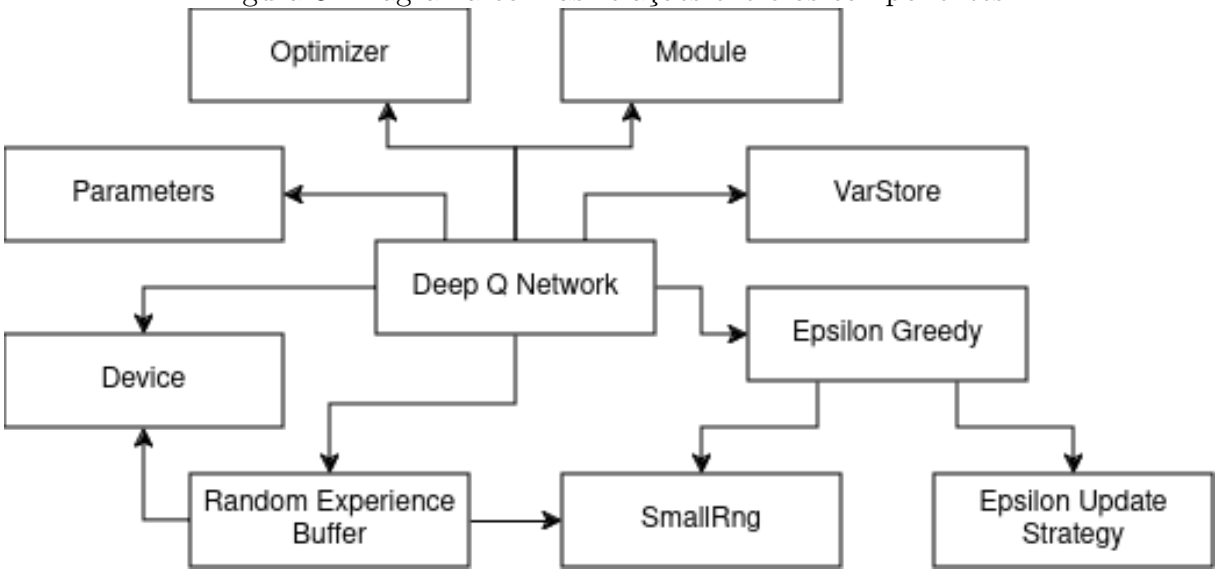


Foi utilizada a biblioteca *tch-rs* que fornece *bindings* para a biblioteca *torch* para treinamento de redes neurais, implementada em *C++* a qual também é o backend da biblioteca *PyTorch* utilizada pela implementação da *Stable Baselines 3*.

Os atributos *Policy* e *Target Policy* da *struct Deep Q Network* representam as duas redes utilizadas pelo algoritmo *DQN*, enquanto que o *Random Experience Buffer* representa o *replay buffer*. Já os demais componentes: *Epsilon Update Strategy*, *Epsilon Greedy* e *Parameters* são utilizados para auxiliar a organização da implementação, sendo responsáveis respectivamente, por: controlar a forma de atualização da taxa de exploração, controlar a escolha entre explorar novos estados ou utilizar o conhecimento já obtido durante o treinamento e armazenar os parâmetros de configuração do algoritmo *DQN*.

As relações entre os componentes previamente apresentados podem ser melhor visualizadas na Figura 8 a seguir:

Figura 8: Diagrama com as relações entre os componentes



Dessa forma, na inicialização de uma *struct Deep Q Network* deve-se fornecer uma instância do *Epsilon Greedy*, do *Random Experience Buffer*, uma função que gere as redes neurais via a abstração de *Modules* e *VarStores*, um *Optimizer*, a *loss_fn*, e por fim, os parâmetros do algoritmo via a *struct Parameters* e o *Device* em que o treinamento ocorrerá, isto é, na *CPU* ou na *GPU*.

Porém, tal inicialização só é necessária na utilização da implementação em *Rust*. Com os *bindings* para *Python*, o uso do algoritmo fica mais simples. Os *bindings* foram feitos por meio de um *wrapper* via a biblioteca *PyO3/Maturin*, conforme o exemplo de código apresentado na Figura 9, necessário para sua utilização em *Python*.

Figura 9: Uso dos *bindings* para *Python* do algoritmo *DQN* implementado em *Rust*

```
1  model = DQN([(256, "relu"), (256, "relu")], 0.0023)
2  train_results = model.train(env, eval_env, step=50_000)
3  test_results = model.evaluate(eval_env, 10)
```

Toda a implementação realizada está disponível online, tanto o código em *Rust* quanto os *bindings*. A seguir no Algoritmo 1 está o pseudo código da versão do algoritmo *DQN* implementado, baseada na implementação presente na *stable baselines3* com algumas modificações para *Rust*.

Algoritmo 1: Deep Q-learning implementado

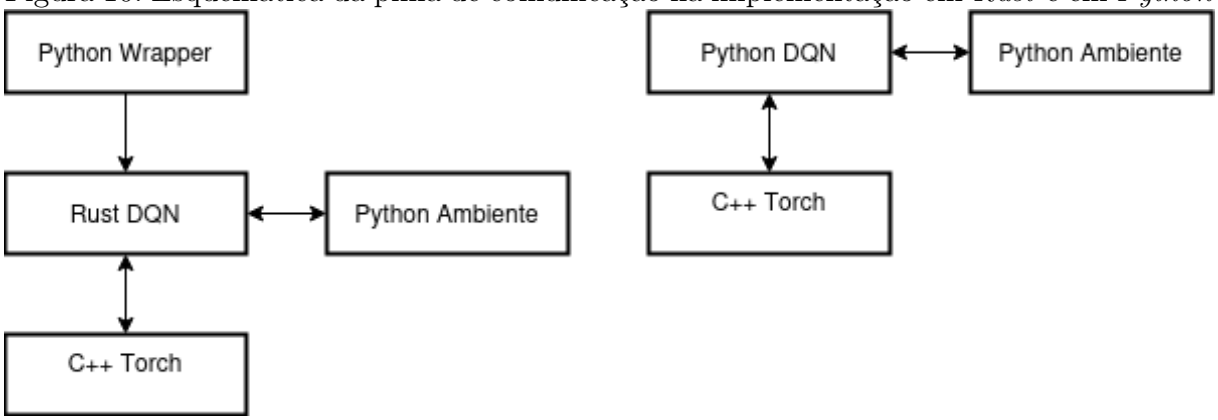
1 Initialize action selector A with episilon ϵ and strategy T
2 Initialize replay memory D to capacity N
3 Initialize action-value function Q with random weights θ
4 Initialize target action-value function $\bar{Q}$ with random weights $\bar{\theta} = \theta$
5 Set $loss\_fn$ to one of: MAE, MSE, RMSE, Huber or Smooth L1
6 Get initial state s_1 from enviroment
7 **for** *t = 1, steps* **do**
8 Get action a_t from A based on s_t
9 Execute action a_t and get reward r_t, done d_t and next observation s_{t+1}
10 Store transition $(s_t, a_t, r_t, d_t, s_{t+1})$ in D
11 Every T steps **do for** *g = 1, gradient steps* **do**
12 Sample random mini-batch of transitions $(s_j, a_j, r_j, d_j, s_{j+1})$ from D
13 Set $y_j = \begin{cases} r_j & \text{when } d_j \text{ True} \\ r_j + \gamma \max_{a'} Q(s_{j+1}, a'; \bar{\theta}) & \text{when } d_j \text{ False} \end{cases}$
14 Perform a gradient descent step with $loss\_fn(Q(s_j, a_j; \theta), y_j)$ on θ
15 **end**
16 **if** *d_t is True* **then**
17 Start new episode, get new s_{t+1} from enviroment
18 Update A with new ϵ following strategy T
19 Every C episodes **do** set $\bar{\theta} = \theta$
20 **end**
21 Every E steps **do** evaluate agent on 10 episodes and get $mean\_reward$
22 **if** *$mean\_reward$ >= environment threshold reward* **then**
23 Stop training
24 **end**
25 **end**

As principais alterações consistem na adição de uma frequência de treinamento dependente do passo atual, e na quantidade de atualizações dos pesos da rede, ambos na linha 11, além do mecanismo de interrupção do treinamento da linha 22. Além da verificação para interromper o treinamento nas linhas 21-23.

Tais alterações também estão presentes na implementação da *Stable Baselines3*, porém foram implementadas de formas diferentes, pois a biblioteca em questão dá suporte a outras configurações e algoritmos usando a mesma estrutura de treinamento.

Por fim, sobre a implementação, vale destacar como ficou a *stack* de tecnologias e a comunicação entre elas, ilustrada na Figura 10:

Figura 10: Esquemática da pilha de comunicação na implementação em *Rust* e em *Python*



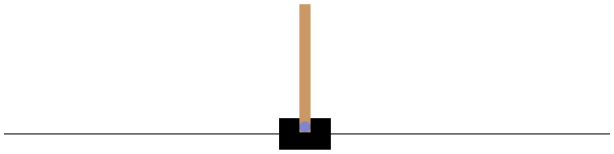
Anteriormente tanto a execução do ambiente quanto a do algoritmo de aprendizado ocorriam em *Python*, apenas o treinamento das redes que era otimizado pela biblioteca *Torch*. Enquanto que com a abordagem utilizada, o ambiente continua sendo executado em Python, o que acaba gerando certo *overhead*, porém as instruções do algoritmo DQN ocorrem em *Rust* e a comunicação com a biblioteca *Torch* também passou a ser com *Rust*, o que reduz os custos de *marshalling* dos dados e permite ainda mais velocidade. Enquanto que devido ao *wrapper* em *Python* o usuário não sente tal alteração, visto que continua configurando o algoritmo em *Python*.

Na seção seguinte será exposto sobre o que é o ambiente, qual seu objetivo e suas restrições.

4.2 O ambiente *Cart Pole*

O ambiente do *Cart Pole* é um *benchmark* clássico para problemas de controle na área do aprendizado por reforço. Neste ambiente, o agente é responsável por equilibrar um poste que é articulado a um carrinho que se move horizontalmente aplicando forças à base do carrinho. Destaca-se ainda, que as equações de movimento do sistema carrinho-poste não são conhecidas e que o único *feedback* é um sinal positivo que é emitido enquanto o agente não falhar, a falha ocorrendo quando o poste cai além de um certo ângulo da vertical, ou quando o carrinho atinge o fim da pista (BARTO; SUTTON; ANDERSON, 1983).

Figura 11: Representação gráfica do ambiente *Cart Pole*



FONTE: *Gymnasium* - Biblioteca *Python* de ambientes para RL

Portanto, o agente interage com o ambiente indicando a cada passo uma de duas ações: se locomover para a esquerda ou se locomover para a direita. Sendo tais ações representadas pelos valores 0 ou 1, respectivamente.

É importante destacar que a velocidade que é reduzida ou aumentada pela força aplicada em cada ação não é fixa e depende do ângulo que o poste está apontando. O centro de gravidade do poste varia a quantidade de energia necessária para mover o carrinho abaixo dele, afetando portanto a velocidade de locomoção.

A observação que o agente dispõe para tomar suas ações a cada passo é composta de 4 valores reais: a posição do carrinho, a velocidade do carrinho, o angulo do poste e a velocidade angular do poste. Os intervalos de valores possíveis das variáveis que compõem a observação são mostrados na Tabela 4 abaixo.

Tabela 4: Intervalos dos valores que compõem uma observação do ambiente *Cart Pole*

Observação	**Valor mínimo**	**Valor máximo**
Posição do carrinho	-4.8	4.8
Velocidade do carrinho	-∞	∞
Ângulo do poste	∼-0.418 rad (-24°)	∼0.418 rad (24°)
Velocidade angular do poste	-∞	∞

As condições de termino do episódio, isto é, que indicam o fim da emissão do sinal de recompensa, estão descritas na Tabela 5 a seguir.

Tabela 5: Intervalos válidos antes que um episódio acabe no ambiente *Cart Pole*

Observação	**Valor mínimo**	**Valor máximo**
Posição do carrinho	-2.4	2.4
Ângulo do poste	∼-0.2095 rad (-12°)	∼0.2095 rad (12°)

Dessa forma, o ambiente é considerado resolvido por um agente, quando tal agente

obtém um recompensa acumulada acima de 475 ao fim de um episódio. Assim, por questões de economia de tempo, os episódios são limitados a 500 passos de interação.

Na inicialização do ambiente, todos os 4 valores que compõem a observação são escolhidos uniformemente entre o intervalo (-0.05, 0.05). Assim, é possível indicar uma *seed* para reprodutibilidade do mesmo estado inicial.

Para mais detalhes sobre o ambiente, consultar a documentação oficial da biblioteca *Gymnasium* e também o código-fonte da implementação do ambiente.

4.3 Metodologia Experimental

Os experimentos realizados consistiram em coletar tempos de 100 execuções da implementação do algoritmo *DQN* em *Rust* com *bindings* para *Python* no ambiente do *Cart Pole*, bem como outras 100 execuções do algoritmo *DQN* em *Python* da biblioteca *Stable Baselines 3* no mesmo ambiente.

Em cada execução dos algoritmos foi utilizada uma *seed* diferente, variando incrementalmente de 0 até 99, de forma a reproduzir os mesmos resultados quando necessário. O treinamento do algoritmo é interrompido se a recompensa mínima para considerar o ambiente resolvido for atingida. Essa condição é verificada a cada 1000 passos pela média de 10 execuções em uma instância de avaliação do ambiente.

4.3.1 Descrição dos parâmetros usados

Foram utilizados dois conjuntos de parâmetros, cada um otimizado para a devida implementação. Os parâmetros usados para a versão da *SB3* são os recomendados no repositório da *SB3 Zoo*, contém os parâmetros já ajustados para múltiplos algoritmos em diferentes ambientes.

Os parâmetros usados na implementação em *Rust* também se basearam nos recomendados na *SB3 Zoo*, porém ajustados de forma exploratória, visto que inicialmente não permitiam a convergência do algoritmo para a solução do ambiente. Após uma investigação, acredita-se que tais diferenças se deram por questões de precisão da representação dos *floats*, no momento da conversão dos valores entre *Python* e *C++ (torch)* ocorre uma perda de precisão devido a forma de representação que cada linguagem segue, mesmo implementando o padrão *IEEE 754*, enquanto que na versão em *Rust* tal perda não ocorre, visto que a representação interna do *float* é a mesma entre *Rust* e *C++*.

Assim, foi constatado que nas iterações iniciais ambas as implementações se comportam da mesma forma, porém após algumas iterações de atualização dos pesos das redes, observa-se pequenas divergências nos valores para cada ação, até que em determinado momento, ações diferentes são escolhidas por cada implementação. Após essa diferença da ação escolhida os caminhos de cada implementação divergem completamente.

Dessa forma, devido a tal limitação, foi necessário usar parâmetros diferentes para cada implementação, de forma a cobrir tanto uma parametrização otimizada para o *SB3* quanto para o *DQN* implementado.

Na Tabela 6 estão organizadas breves descrições de cada hiper-parâmetro do algoritmo DQN presentes tanto na implementação feita em *Rust* quanto na implementação presente na *Stable Baselines3*.

Tabela 6: Descrição dos parâmetros utilizados no algoritmo DQN

Parâmetro	**Descrição**
network_arch	arquitetura de rede usada, (neurônio por camada)
hidden_activation	funções de ativação entre camadas
learning_rate	taxa de aprendizado para otimizador
memory_size	tamanho do *buffer* de repetição
min_memory_size	número de entradas para iniciar o treinamento
discount_factor	fator de desconto para recompensas futuras (γ)
initial_epsilon	valor inicial do epsilon para exploração
final_epsilon	valor final do epsilon para exploração mínima
exploration_fraction	fração do treinamento em que o epsilon decresce
max_grad_norm	limite para gradientes durante atualizações de parâmetros
optimizer	algoritmo de otimização para ajustar pesos de rede
normalization	indica se deve normalizar as observações
loss_function	métrica para avaliar a perda
steps	interações totais com o ambiente durante o treinamento
gradient_steps	atualizações de parâmetros após cada interação
train_freq	interações antes de uma atualização de parâmetro
update_freq	frequência de atualização da rede de destino
batch_size	número de amostras usadas para atualizar os parâmetros
eval_freq	intervalo entre avaliações de desempenho do agente
eval_for	número de episódios ou interações para avaliar o desempenho

Enquanto que na Tabela 7 estão os valores usados para cada hiper-parâmetro em cada implementação.

Tabela 7: Hiper-parâmetros usados para os experimentos com o DQN

Parâmetro	**SB3**	**Rust**	**Parâmetro**	**SB3**	**Rust**
network arch	[256,256]	[256,256]	optimizer	ADAM	ADAM
hidden activation	[relu,relu]	[relu,relu]	normalization	yes	no
learning rate	0.0023	0.01	loss function	Huber	Huber
memory size	100000	10000	steps	50000	50000
min memory size	1000	1000	gradient_steps	128	2
discount factor	0.99	0.99	train_freq	256	2
initial epsilon	1.0	0.5	update_freq	10	2
final epsilon	0.04	0.05	batch_size	64	256
exploration fraction	0.16	0.2	eval_freq	1000	1000
max grad norm	10	10	eval_for	10	10

É importante mencionar que a parametrização selecionada para o *DQN* implementado foi testada no *DQN* da *SB3*, porém o resultado é que o algoritmo não converge para a solução esperada do ambiente. Os valores foram selecionados após execução da ferramenta *Optuna* para otimização de hiper parâmetros. Destacando que a não normalização das observações se deu por uma limitação ao interagir com o ambiente, visto que ele executava em *Python*, na SB3 é utilizado um *wrapper* no ambiente que com base na interação estima a média e o desvio padrão, e no momento de amostrar os dados do *replay buffer* o ambiente é usado para normalizar as amostras selecionadas. Utilizar essa mesma abordagem em Rust teria um custo muito elevado, pois aumentaria consideravelmente a quantidade de invocações aos *bindings* para *Python*.

4.3.2 Configuração de *Hardware* & *Software*

Foram utilizadas as instalações do Núcleo de Processamento de Alto Desempenho (NPAD) da UFRN, com os seguintes *hardwares* e *softwares*

- 1 CPU: Intel(R) Xeon(R) CPU E5-2683 v4 @ 2.10GHz/40M cache/ 9.6 GT/s QPI
- 1 *GPU*: Tesla V100-SXM2-16GB/ 5,210 GPU Cores/ 4 X DP 1.4
- Sistema Operacional: Rocky Linux 8.5 (Green Obsidian)
- Versão da linguagem Rust: 1.77.2
- Versão da linguagem Python: 3.11.4

- Versão do Torch: 2.0.1
- Versão da SB3: 2.3.1
- Versão do PyO3/Maturin: 0.21.2/1.5.1

4.3.3 Métricas utilizadas na comparação

As métricas observadas nos experimentos estão descritas na Tabela 8 abaixo.

Tabela 8: Descrição das métricas de avaliação das implementações do algoritmo *DQN*

Métrica	**Descrição**
tempo de treinamento (segundos)	Tempo total em segundos que o algoritmo levou para convergir ou alcançar o limite de passos estabelecido
passos executados no treinamento	Quantidade total de passos de interação com o ambiente que o agente executou durante seu treinamento
memória máxima (megabytes)	Quantidade máxima de memória usada durante a execução do algoritmo, medida em megabytes
média de recompensas (10 episódios)	Média de recompensas obtidas pelo algoritmo em 10 episódios após o fim do seu treinamento

O tempo de treinamento em segundos é uma das principais métricas que se deseja avaliar, pois reduzir o tempo de treinamento permite iterações mais rápidas para o desenvolvimento de aplicações que fazem uso desses algoritmos. Isso permite que mais trabalho seja feito, reduzindo o tempo necessário para se obter um modelo treinado para resolver determinado problema.

A quantidade de passos executados no treinamento é um indicador relevante pois mostra o custo que se tem com a simulação, sendo outra forma de avaliar o quão rápido o algoritmo treina. Pois quanto menos interação houver com o ambiente, principalmente ambientes custosos de se atuar, melhor será para o pesquisador ou desenvolvedor.

A memória máxima utilizada é uma métrica que permite julgar outro custo importante dos modelos, pois se muita memória for necessária para sua execução, pode acabar

inviabilizando sua utilização em determinadas aplicações. Idealmente, quanto menor for o uso de memória, melhor será pois permitirá baratear a sua utilização.

Por fim, a média de recompensas ao final dos episódios é uma métrica para a qualidade dos resultados obtidos. Pois não basta treinar rapidamente, uma vez que a qualidade e a robustez do modelo após o treinamento são fundamentais para a utilização dos algoritmos.

A partir dessas métricas é avaliada também a razão entre tempo de treinamento e quantidade de passos utilizada. Assim, é possível estimar a eficiência do algoritmo por passo executado e, ao dividir o tempo gasto pela quantidade de passos utilizada, obtemos um valor que indica quanto tempo o algoritmo gasta por passo.

4.3.4 Forma de avaliação dos resultados

Para a valiar os resultados obtidos utilizando as métricas selecionadas, serão comparadas as médias, desvio padrões, valores mínimos e valores máximos entre todas as 100 execuções dos algoritmos. De forma que, em média, um menor tempo de execução do treinamento, uma menor quantidade de passos para treinamento, uma menor razão entre tempo de treinamento e quantidade de passos de treinamento, uma menor utilização de memória e uma maior recompensa após treinamento sejam obtidas. Sendo que é preferível um menor desvio padrão nessas 100 execuções, para cada uma dessas métricas, pois indica maior consistência nos resultados.

Para validar a significância estatística dos resultados são utilizados três testes estatísticos. Portanto, inicialmente, formulam-se duas hipóteses. A hipótese nula (H_0) sugere que não há diferença significativa entre as médias dos dois grupos, enquanto a hipótese alternativa (H_1) sugere que há uma diferença significativa.

O primeiro teste é o Teste T de Student (STUDENT, 1908), que é amplamente utilizado para determinar se há uma diferença significativa entre as médias de dois grupos independentes ou relacionados. Ele se baseia na suposição de que os dados seguem uma distribuição normal e que as variâncias dos dois grupos são iguais/similares.

O valor do teste T é calculado utilizando a diferença entre as médias dos grupos, a variância dentro de cada grupo e o tamanho da amostra. A fórmula para o teste T de duas amostras independentes é:

$$t = \frac{\bar{x}_1 - \bar{x}_2}{\sqrt{s_p(\frac{1}{n_1} + \frac{1}{n_2})}} \tag{4.1}$$

onde $\bar{x}_1$ e $\bar{x}_2$ são as médias dos dois grupos, n_1 e n_2 são os tamanhos dos grupos e s_p é calculado da seguinte forma, com $s^2_{x_1}$ e $s^2_{x_2}$ sendo as variâncias de cada grupo:

$$s_p = \sqrt{\frac{(n_1 - 1)s^2_{x_1} + (n_2 - 1)s^2_{x_2}}{n_1 + n_2 - 2}} \tag{4.2}$$

O segundo teste é o Teste T de Welch (WELCH, 1947). Uma versão modificada do teste T de Student, empregada para comparar as médias de dois grupos independentes quando as variâncias dos grupos são diferentes e/ou os tamanhos das amostras são desiguais. Este teste é mais robusto contra a violação da suposição de homogeneidade de variâncias, oferecendo uma análise estatística mais precisa sob essas condições.

O valor do teste T de Welch é calculado usando uma fórmula que ajusta para diferenças nas variâncias e tamanhos das amostras:

$$t = \frac{\bar{x}_1 - \bar{x}_2}{\sqrt{\frac{s^2_{x_1}}{n_1} + \frac{s^2_{x_2}}{n_2}}} \tag{4.3}$$

O terceiro teste é o Teste U de Man-Whitney (MANN; WHITNEY, 1947) também conhecido como teste de Mann-Whitney-Wilcoxon. É uma alternativa ao teste T de Student quando as suposições de normalidade e homogeneidade de variâncias não são atendidas.

Para calcular o valor do Teste U de Man-Whitney, se faz:

- Os dados de ambas as amostras são combinados e ordenados em uma única sequência crescente;
- Cada valor é substituído pelo seu posto (rank) na sequência ordenada. Em caso de empates, é utilizado o posto médio dos valores empatados;
- U é calculado a partir da soma dos postos dos valores de cada grupo

$$U_1 = n_1 n_2 + \frac{n_1(n_1 + 1)}{2} + R_1 \tag{4.4}$$

$$U_2 = n_1 n_2 + \frac{n_2(n_2 + 1)}{2} + R_2 \tag{4.5}$$

  onde n_1 e n_2 são os tamanhos das duas amostra. Enquanto que R_1 e R_2 são as somas dos postos dos valores dos grupos 1 e 2, respectivamente
- o U utilizado é o menor valor entre U_1 e U_2.

Após aplicar os três testes apresentados, compara-se o *p-value* ao nível de significância (α) escolhido que foi de $0,05$. Se o *p-value* for menor que α, rejeita-se a hipótese nula, concluindo que há uma diferença significativa entre as médias dos grupos. Caso contrário, não se rejeita a hipótese nula.

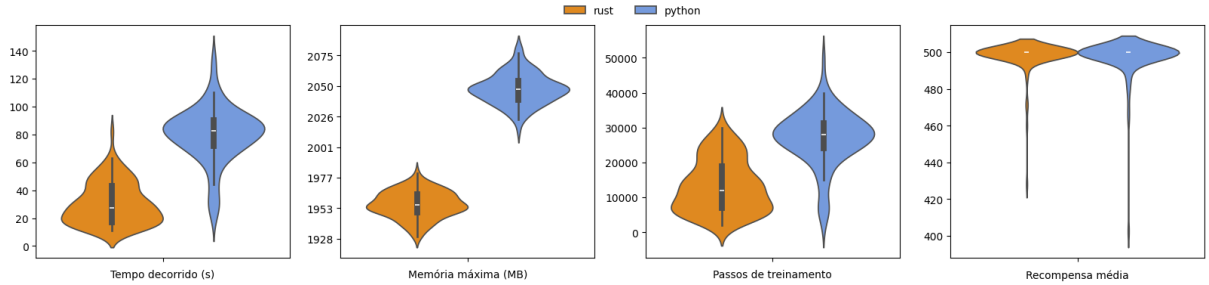
5 Resultados e Discussões

Neste capítulo, os resultados dos experimentos descritos no capítulo anterior são apresentados e discutidos. Inicialmente as distribuições dos resultados para cada uma das métricas avaliadas são apresentadas, seguidas das estatísticas descritivas dessas distribuições. Logo mais, ocorre a comparação em si desses resultados, discutindo e avaliando segundo os critérios descritos no capítulo anterior. Por fim, os resultados dos testes estatísticos são exibidos e comentados.

5.1 Comparação dos resultados

As distribuições dos resultados para cada uma das métricas avaliadas podem ser visualizadas na Figura 12 abaixo, sendo óbvias as diferenças para o tempo decorrido no treinamento, memória consumida e quantidade de passos de treinamento. Enquanto que para a recompensa média já não é perceptível alguma diferença óbvia entre os resultados. Destaca-se o uso de diferentes escalas para cada sub-figura, pois o foco dessa análise inicial é na forma das distribuições e na comparação entre elas para cada variável.

Figura 12: Gráficos de violino para cada uma das quatro métricas analisadas



Apenas observando as distribuições já é possível afirmar que não há diminuição na qualidade das soluções obtidas com a versão *Rust* em comparação com a versão da *SB3*. Ao mesmo tempo que há uma diminuição no tempo de treinamento, quantidade de memória

utilizada e na quantidade de passos de interação com o ambiente. Porém ainda se faz necessário analisar estatisticamente se os resultados de fato são significativos.

Na Tabela 9 estão algumas informações descritivas sobre os resultados obtidos com o experimento, contendo média, desvio padrão, valor mínimo e valor máximo:

Tabela 9: Estatística descritiva dos valores das métricas obtidas com o experimento

Métrica	**Info**	**SB3**	**Rust**	**Métrica**	**Info**	**SB3**	**Rust**
Tempo de treinamento em segundos	mean	80.14	30.45	Memória máxima em megabytes	mean	2097.65	1956.15
	std	19.96	14.82		std	12.60	10.60
	min	19.36	10.81		min	2063.81	1930.30
	max	134.97	82.15		max	2131.12	1981.32
Quantidade de Passos durante o treinamento	mean	27140	13220	Média das Recompensas	mean	497.33	497.88
	std	7960.79	7353.39		std	11.26	9.20
	min	2000	2000		min	402.70	428.20
	max	50000	30000		max	500.00	500.00

Como se pode observar, a implementação em *Rust* foi em média cerca de 50 segundos mais rápida do que a implementação da *SB3*, o que equivale a uma redução de 62,5%, e com um desvio padrão aproximadamente 34% menor. Enquanto que o número médio de passos de treinamento foi um pouco menor que a metade da *SB3*, com um desvio padrão aproximadamente 18% menor. Tais diferenças se dão tanto devido a implementação do algoritmo quanto à parametrização utilizada, afinal, devido as limitações previamente mencionadas, infelizmente não foi possível a utilização dos mesmos hiper parâmetros para os dois experimentos.

Já em relação ao consumo de memória, a versão em *Rust* consumiu em média 141.5 *megabytes* a menos que a versão da *SB3*, o que não é uma quantidade significativa, sendo aproximadamente apenas 7% de redução. Já a recompensa média após o treinamento teve uma média muito próxima da versão da *SB3*. Portanto, os resultados após o treinamento são praticamente iguais no que tange a efetividade para resolver o problema proposto pelo ambiente.

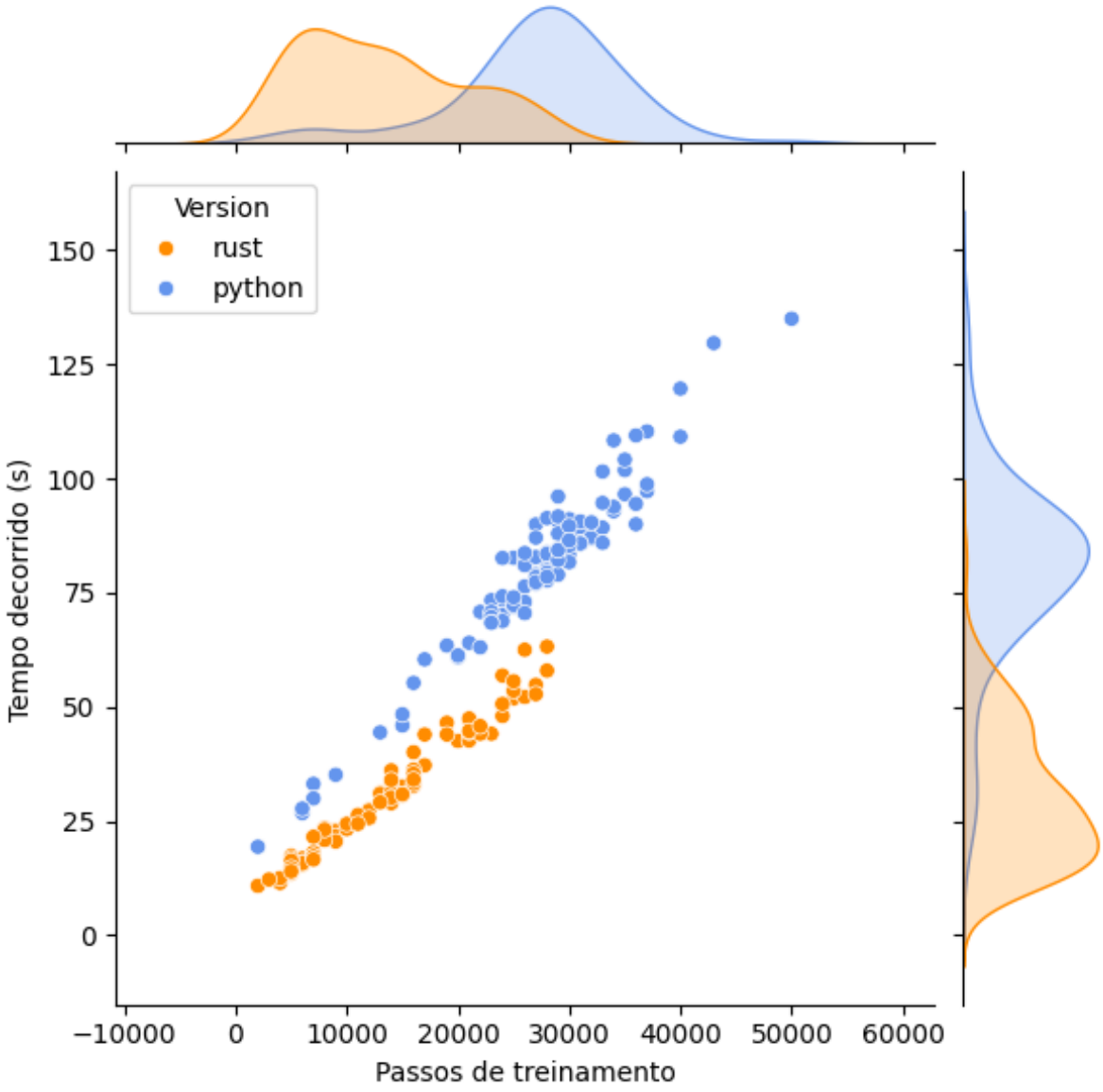
Sobre a razão de tempo por passo, na Tabela 10 estão os valores obtidos para cada implementação, sendo que a versão em *Rust* apresenta valores menores para as quatro informações analisadas. Tal resultado demonstra que mesmo com as diferenças entre parâmetros há uma redução no tempo intrínseca à mudança da linguagem para *Rust*.

Tabela 10: Estatística descritiva da razão tempo por passo de treinamento

Métrica	**Info**	**SB3**	**Rust**
Razão entre tempo por passo de treinamento	mean	0.003095	0.002475
	std	0.000767	0.000484
	min	0.002499	0.001917
	max	0.009680	0.005405

O gráfico da Figura 13 permite uma compreensão visual das diferenças de tempo de cada implementação experimentada, reforçando as afirmações sobre a versão em *Rust* possuir uma menor razão de tempo por passo de treinamento.

Figura 13: Comparação entre as razões tempo/passos das duas versões



Na seção a seguir estão descritos os resultados dos testes estatísticos para validar a relevância dos resultados discutidos nesta seção.

5.2 Testes estatísticos

Na Tabela 11 é apresentado os resultados dos três testes estatísticos realizados

Tabela 11: Resultado dos testes de significância estatística

Métrica	Teste	Estatística	Graus de liberdade	p
Tempo decorrido (s)	Student	-19.986	198.000	<0.001
	Welch	-19.986	182.747	<0.001
	Mann-Whitney	386.000	-	<0.001
Memória Máxima (MB)	Student	-56.825	198.000	<0.001
	Welch	-56.825	193.776	<0.001
	Mann-Whitney	0.000	-	<0.001
Passos de treinamento	Student	-12.845	198.000	<0.001
	Welch	-12.845	196.766	<0.001
	Mann-Whitney	1045.000	-	<0.001
Razão Tempo / Passo	Student	-6.837	198.000	<0.001
	Welch	-6.837	167.030	<0.001
	Mann-Whitney	1260.000	-	<0.001
Média de Recompensa	Student	0.379	198.000	0.647
	Welch	0.379	190.442	0.647
	Mann-Whitney	5104.500	-	0.682

Conforme mostrado na Tabela 11, podemos concluir que há evidências estatísticas para apoiar a afirmação de que a implementação em *Rust* é mais rápida do que a versão *Python*, sem impacto significativo nas recompensas obtidas. Isso se deve ao fato que não há evidências estatísticas para afirmar que as distribuições da média de recompensa sejam diferentes entre as versões.

Os três testes concordam com tais afirmações, mesmo com suas diferenças, sendo o teste de *Mann-Whitney* o mais confiável para o problema, devido às visíveis diferenças entre as distribuições analisadas, pois as distribuições de tempo e passos na implementação em *Rust* não seguem uma distribuição normal.

Portanto, a versão implementada em *Rust* se apresenta como uma alternativa mais rápida que a implementação da *SB3* sem comprometer a qualidade dos resultados obtidos.

6 Considerações finais

Este capítulo resume o que foi desenvolvido no presente trabalho, sintetizando os resultados e suas principais contribuições e listando as limitações do estudo realizado. Por fim, apresentam-se algumas perspectivas para desenvolvimento de trabalhos futuros.

6.1 Síntese do trabalho

Em resumo, o estudo apresenta uma comparação entre uma implementação do algoritmo *DQN* em *Rust* e a implementação em *Python* presente na biblioteca *Stable Baselines*3, ambas sendo executadas no ambiente *Cart Pole*. Inicialmente contextualizou-se o problema do crescente custo que a execução dos algoritmos de aprendizado por reforço vêm apresentando, em especial durante seu treinamento.

Em seguida a motivação da solução proposta foi exposta, sendo devido a outros exemplos de problemas custosos na área de aprendizado de máquina no qual implementações em linguagens com maior performance foram utilizadas. Além de sua justificativa, isto é, qual o motivo pelo qual uma implementação em *Rust* foi escolhida ao invés de alguma outra linguagem. Essa escolha se deu devido a Rust ter seu desempenho similar a *C*, mas com segurança de memória sem uso de *garbage collector*, assim como também por possuir um ambiente de desenvolvimento mais moderno do que *C/C++*, principalmente no que tange a resolução de dependências.

Em seguida o objetivo geral do estudo foi exposto: Verificar empiricamente se uma implementação em *Rust* do algoritmo *Deep Q-Network* é mais rápida do que uma implementação em *Python*. Assim como os objetivos específicos:

1. Implementar em *Rust* o algoritmo *DQN* e também seus *bindings* para *Python*, visando facilitar sua integração com outras bibliotecas em *Python*;
2. Experimentar as versões em *Rust* e *Python* do *DQN* no ambiente do *Cart-Pole*;

3. Realizar uma análise comparativa dos resultados dos experimentos;

Após ter apresentado como o algoritmo *DQN* foi implementado e o funcionamento do ambiente do *Cart Pole*, onde os experimentos foram executados, os hiper-parâmetros utilizados foram informados, bem como seus valores para as duas implementações comparadas. Em seguida foram apresentadas as configurações de *hardware* e *software*, assim como as métricas utilizadas na comparação, sendo elas: tempo de treinamento em segundos; quantidade de passos de treinamento; memória máxima utilizada em megabytes; e média de recompensas após treinamento. Além de uma razão entre o tempo de treinamento e a quantidade de passos de treinamento. Por fim, foi indicado como os resultados foram avaliados, fazendo uso também dos testes estatísticos de *Student*, *Welch* e *Mann-Whitney*.

Ao comparar as distribuições nas quatro métricas avaliadas, bem como as medidas descritivas de cada um delas, constata-se que a versão em *Rust* foi mais rápida de treinar, utilizou menos passos e menos memória e alcançou resultados tão bons quanto a versão da *SB3* em *Python*, além de manter também uma menor razão de tempo por passos de treinamento.

Por último os resultados dos testes estatísticos foram exibidos e o veredito final sobre a comparação foi comprovado. De fato a implementação em *Rust* se apresenta mais rápida que a da *SB3*, tendo uma comprovação estatística, principalmente ao considerar o teste de *Mann-Whitney* mais adequado quando as distribuições em questão não são normais.

Destaca-se que os objetivos específicos foram alcançados ao fim do trabalho, foi implementado o algoritmo *DQN* em *Rust*, bem como seus *bindings* para *Python*, foram realizados experimentos usando o ambiente *Cart Pole* entre a implementação em *Rust* e uma implementação em *Python*. Assim como foi feita uma análise dos resultados desses experimentos, onde foi verificado empiricamente que uma implementação em *Rust* do algoritmo *DQN* é mais rápida do que uma implementação em *Python*, em especial do que a implementação disponível na *Stable Baselines3*.

6.2 Limitações e Trabalhos Futuros

A principal limitação do trabalho se dá na utilização de apenas um ambiente de teste. Além de ser um ambiente simples de ser simulado, também não levada em conta o uso de vetorização do ambiente como forma de aumentar o desempenho do treinamento.

Outra limitação é na quantidade de algoritmos utilizados, pois apenas o *DQN* foi comparado e ainda mais com apenas uma possível implementação em *Python*, o que não exclui a possibilidade de outras implementações serem mais rápidas.

Por fim, outra limitação relevante do estudo consiste no fato de não ter sido explorado o paralelismo. Técnica muito relevante para obter aumento de performance e com isso a diminuição do tempo de treinamento dos algoritmos. Ainda mais levando em consideração que a linguagem *Rust* oferece *fearless concurreny* (concorrência sem medo), pois as propriedades de posse e de empréstimo dos dados ajudam a evitar armadilhas comuns do desenvolvimento de algoritmos concorrentes e/ou paralelos.

Desse modo, diante das limitações do estudo, propõe-se como trabalhos futuros para melhorar o trabalho, as seguintes possibilidades:

- Realizar experimentos em outros ambientes clássicos como o *Pendulum* e o *Acrobot*;
- Realizar experimentos em ambientes modernos: *Atari*, *MuJoCo* ou *StarCraft II*;
- Realizar experimentos em ambientes vetorizados, permitindo maior desempenho na interação com o ambiente, prática que é cada vez mais uma tendência na área;
- Implementar e comparar outros algoritmos além do *DQN*, como o *PPO* ou o *TD3*;
- Comparar o algoritmo com outras implementações e em outras linguagens;
- Implementar e comparar a versão paralela do algoritmo *DQN* e também de outros algoritmos como o *PPO* ou o *TD3*, potencialmente distribuindo a computação;
- Comparar o desempenho das implementações em hardwares mais modestos;
- Analisar os custos energéticos entre as implementações;

Referências

AHO, A. V. et al. *Compilers*. 2. ed. Upper Saddle River, NJ: Pearson, 2006.

ANDERSON, J. *Python Bindings: Calling C or C++ From Python*. mar. 2020. Disponível em: <<https://realpython.com/python-bindings-overview>>. Acesso em Dezembro 17, 2023.

BARTO, A. G.; SUTTON, R. S.; ANDERSON, C. W. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13, n. 5, p. 834–846, 1983.

BROCKMAN, G. et al. *OpenAI Gym*. arXiv, 2016. Disponível em: <https://arxiv.org/abs/1606.01540>.

BUGDEN, W.; ALAHMAR, A. Rust: The programming language for safety and performance. *2nd International Graduate Studies Congress (IGSCONG'22)*, arXiv, 2022. Disponível em: <https://arxiv.org/abs/2206.05503>.

CRICHTON, A. *Rust Once, Run Everywhere*. abr. 2015. Disponível em: <<https://blog.rust-lang.org/2015/04/24/Rust-Once-Run-Everywhere.html>>. Acesso em Dezembro 17, 2023.

DONG, H.; DING, Z.; ZHANG, S. *Deep Reinforcement Learning: Fundamentals, Research and Applications*. Springer Singapore, 2020. ISBN 9789811540950. Disponível em: <http://dx.doi.org/10.1007/978-981-15-4095-0>.

DU, Y.-M. et al. Reinforcement learning method based on radial basis function neural network. In: *Design, Manufacturing and Mechatronics*. [S.l.]: WORLD SCIENTIFIC, 2017.

EMERY, D. *Standards, APIs, Interfaces and Bindings*. dez. 1996. Disponível em: <<http://oldwww.acm.org/tsc/apis.html>>. Acesso em Dezembro 17, 2023.

ERNEST, A.; MENSAH, E.; GILBERT, A. Qualitative assessment of compiled, interpreted and hybrid programming languages. *Communications on Applied Electronics*, Foundation of Computer Science, v. 7, n. 7, p. 8–13, out. 2017. ISSN 2394-4714. Disponível em: <http://dx.doi.org/10.5120/cae2017652685>.

ESCHMANN, J.; ALBANI, D.; LOIANNO, G. *RLtools: A Fast, Portable Deep Reinforcement Learning Library for Continuous Control*. [S.l.]: arXiv, 2023.

FAWZI, A. et al. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, Springer Science and Business Media LLC, v. 610, n. 7930, p. 47–53, out. 2022. ISSN 1476-4687. Disponível em: <http://dx.doi.org/10.1038/s41586-022-05172-4>.

GIBNEY, E. Self-taught ai is best yet at strategy game go. *Nature*, Springer Science and Business Media LLC, out. 2017. ISSN 1476-4687. Disponível em: <http://dx.doi.org/10.1038/nature.2017.22858>.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. *Deep Learning*. 1. ed. [S.l.]: MIT Press, 2016. <http://www.deeplearningbook.org>. ISBN 0262035618.

HUANG, S. et al. Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms. *Journal of Machine Learning Research*, v. 23, n. 274, p. 1–18, 2022. Disponível em: <http://jmlr.org/papers/v23/21-1342.html>.

HUNGER, T. *Rust and C++ Interoperability*. nov. 2022. Disponível em: <<https://slint.dev/blog/rust-and-cpp>>. Acesso em Dezembro 17, 2023.

JIN, C. et al. *Provably Efficient Reinforcement Learning with Linear Function Approximation*. [S.l.]: arXiv, 2019.

JONES, R.; HOSKING, A.; MOSS, E. *The garbage collection handbook*. 2. ed. Philadelphia, PA: Chapman & Hall/CRC, 2023. ("International Perspectives on Science, Culture and Society").

KLABNIK, S.; NICHOLS, C. *The rust programming language*. San Francisco, CA: No Starch Press, 2018. ISBN 1593278284.

KNUTH, D. E. *The art of computer programming*. 3. ed. Boston, MA: Addison Wesley, 1997. ISBN 0201896850.

LECUN, Y.; BENGIO, Y.; HINTON, G. Deep learning. *nature*, Nature Publishing Group UK London, v. 521, n. 7553, p. 436–444, 2015.

LIANG, E. et al. RLlib: Abstractions for distributed reinforcement learning. In: DY, J.; KRAUSE, A. (Ed.). *Proceedings of the 35th International Conference on Machine Learning*. PMLR, 2018. (Proceedings of Machine Learning Research, v. 80), p. 3053–3062. Disponível em: <https://proceedings.mlr.press/v80/liang18b.html>.

MAKOVIYCHUK, V. et al. *Isaac Gym: High Performance GPU-Based Physics Simulation For Robot Learning*. [S.l.]: arXiv, 2021.

MANKOWITZ, D. J. et al. Faster sorting algorithms discovered using deep reinforcement learning. *Nature*, Springer Science and Business Media LLC, v. 618, n. 7964, p. 257–263, jun. 2023. ISSN 1476-4687. Disponível em: <http://dx.doi.org/10.1038/s41586-023-06004-9>.

MANN, H.; WHITNEY, D. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics*, Institute of Mathematical Statistics, v. 18, n. 1, p. 50 – 60, 1947. Disponível em: <https://doi.org/10.1214/aoms/1177730491>.

MEEK, B. Problems of language bindings. *Computer Standards & Interfaces*, Elsevier BV, v. 15, n. 4, p. 353–360, set. 1993. ISSN 0920-5489. Disponível em: <http://dx.doi.org/10.1016/0920-5489(93)90035-p>.

MEYERS, S. *Effective Modern C++*. Sebastopol, CA: O'Reilly Media, 2014.

MITCHELL, T. *Machine Learning*. New York, NY: McGraw-Hill Professional, 1997. (McGraw-Hill series in computer science).

MNIH, V. et al. Playing atari with deep reinforcement learning. arXiv, 2013. Disponível em: <https://arxiv.org/abs/1312.5602>.

MNIH, V. et al. Human-level control through deep reinforcement learning. *Nature*, Nature Publishing Group, a division of Macmillan Publishers Limited. All Rights Reserved., v. 518, n. 7540, p. 529–533, fev. 2015. ISSN 00280836. Disponível em: <http://dx.doi.org/10.1038/nature14236>.

MORAVčíK, M. et al. Deepstack: Expert-level artificial intelligence in heads-up no-limit poker. *Science*, American Association for the Advancement of Science (AAAS), v. 356, n. 6337, p. 508–513, maio 2017. ISSN 1095-9203. Disponível em: <http://dx.doi.org/10.1126/science.aam6960>.

MOSTAFA, B. et al. Machine and deep learning approaches in genome: Review article. *Alfarama Journal of Basic & Applied Sciences*, 08 2020.

NOVATI, G.; KOUMOUTSAKOS, P. Remember and forget for experience replay. *Proceedings of the 36th International Conference on Machine Learning*, 2019.

NOVATI, G.; MAHADEVAN, L.; KOUMOUTSAKOS, P. Controlled gliding and perching through deep-reinforcement-learning. *Physical Review Fluids*, American Physical Society (APS), v. 4, n. 9, set. 2019. ISSN 2469-990X. Disponível em: <http://dx.doi.org/10.1103/PhysRevFluids.4.093902>.

NOVATI, G. et al. Synchronisation through learning for two self-propelled swimmers. *Bioinspiration & Biomimetics*, IOP Publishing, v. 12, n. 3, p. 036001, mar. 2017. ISSN 1748-3190. Disponível em: <http://dx.doi.org/10.1088/1748-3190/aa6311>.

PEREIRA, R. et al. Energy efficiency across programming languages: how do energy, time, and memory relate? In: *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*. ACM, 2017. (SPLASH '17). Disponível em: <http://dx.doi.org/10.1145/3136014.3136031>.

PYEATT, L. D. Reinforcement learning with decision trees. In: *Applied Informatics*. [s.n.], 2003. Disponível em: <https://api.semanticscholar.org/CorpusID:9177125>.

RAFFIN, A. et al. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, v. 22, n. 268, p. 1–8, 2021. Disponível em: <http://jmlr.org/papers/v22/20-1364.html>.

RUDIN, N. et al. Learning to walk in minutes using massively parallel deep reinforcement learning. In: FAUST, A.; HSU, D.; NEUMANN, G. (Ed.). *Proceedings of the 5th Conference on Robot Learning*. PMLR, 2022. (Proceedings of Machine Learning Research, v. 164), p. 91–100. Disponível em: <https://proceedings.mlr.press/v164/rudin22a.html>.

RUSSELL, S.; NORVIG, P. *Artificial intelligence*. 4. ed. Upper Saddle River, NJ: Pearson, 2020. ISBN 8535237011.

SCHMIDHUBER, J. Deep learning in neural networks: An overview. *Neural Networks*, v. 61, p. 85–117, 2015. ISSN 0893-6080. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0893608014002135>.

SILVER, D. et al. Mastering the game of go without human knowledge. *Nature*, v. 550, p. 354–359, 2017. Disponível em: <https://api.semanticscholar.org/CorpusID:205261034>.

STUDENT. The probable error of a mean. *Biometrika*, JSTOR, v. 6, n. 1, p. 1–25, 1908.

SUTTON, R. S.; BARTO, A. G. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. Disponível em: <http://incompleteideas.net/book/the-book-2nd.html>.

TANENBAUM, A. S.; STEEN, M. V. *Distributed Systems*. [S.l.]: Maarten Van Steen, 2023.

TESAURO, G. Temporal difference learning and td-gammon. *Commun. ACM*, Association for Computing Machinery, New York, NY, USA, v. 38, n. 3, p. 58–68, mar 1995. ISSN 0001-0782.

TOWERS, M. et al. *Gymnasium: A Standard Interface for Reinforcement Learning Environments*. arXiv, 2024. Disponível em: <https://arxiv.org/abs/2407.17032>.

TSITSIKLIS, J. N.; ROY, B. V. An analysis of temporal-difference learning with function approximation. *IEEE Trans. Automat. Control*, v. 42, n. 5, p. 674–690, 1997. ISSN 0018-9286,1558-2523. Disponível em: <https://doi.org/10.1109/9.580874>.

TURNER-TRAURING, I. *The hidden performance overhead of Python C extensions*. jan. 2023. Disponível em: <<https://pythonspeed.com/articles/python-extension-performance/>>. Acesso em Dezembro 17, 2023.

VERMA, S.; NOVATI, G.; KOUMOUTSAKOS, P. Efficient collective swimming by harnessing vortices through deep reinforcement learning. *Proceedings of the National Academy of Sciences*, Proceedings of the National Academy of Sciences, v. 115, n. 23, p. 5849–5854, maio 2018. ISSN 1091-6490. Disponível em: <http://dx.doi.org/10.1073/pnas.1800923115>.

WELCH, B. L. The generalization of 'student's' problem when several different population varlances are involved. *Biometrika*, v. 34, n. 1-2, p. 28–35, 01 1947. ISSN 0006-3444. Disponível em: <https://doi.org/10.1093/biomet/34.1-2.28>.

WENG, J. et al. EnvPool: A highly parallel reinforcement learning environment execution engine. In: KOYEJO, S. et al. (Ed.). *Advances in Neural Information Processing Systems*. [S.l.]: Curran Associates, Inc., 2022. v. 35, p. 22409–22421.

ZHOU, X. et al. Learn to navigate: Cooperative path planning for unmanned surface vehicles using deep reinforcement learning. *IEEE Access*, Institute of Electrical and Electronics Engineers (IEEE), v. 7, p. 165262–165278, 2019. ISSN 2169-3536. Disponível em: <http://dx.doi.org/10.1109/ACCESS.2019.2953326>.