UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

HENRIQUE DE PAULA LOPES

BARBELL: um Framework para Modelagem e Simulação de Ambientes de Aprendizado por Reforço

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação

Orientador: Prof. Dr. Bruno Castro Silva

Porto Alegre
2018

“Le véritable voyage de découverte ne consiste pas à chercher de nouveaux paysages, mais à avoir de nouveaux yeux.”

— MARCEL PROUST

RESUMO

Métodos de aprendizado por reforço tratam de problemas compreendidos por uma subárea da inteligência artificial onde um agente, inserido dentro de um ambiente, tenta encontrar a maneira mais eficaz de resolver um problema através da tomada de uma sequência de ações. Ao executar ações dentro do ambiente, o agente recebe *recompensas*, que são valores numéricos que indicam a qualidade de uma ação tomada em um determinado momento. É somente através do valor total das recompensas recebidas pelas ações tomadas pelo agente e pelas mudanças por elas causas no ambiente que ele deve decidir qual será a próxima ação a ser executada. Diferentemente de outras áreas da inteligência artificial, portanto, onde nas quais o agente recebe, para cada decisão tomada, uma resposta que diz se aquela decisão era a melhor naquele momento, métodos de aprendizado por reforço são recomendados em situações onde decisões devem ser tomadas em sequência, sem que necessariamente fique claro para o agente se uma ação tomada é a mais recomendada em um determinado instante.

Problemas de aprendizado por reforço exigem, portanto, que o agente desenvolva uma solução capaz de encontrar a melhor ação a ser tomada em um dado momento, a fim de maximizar o valor total das recompensas recebidas. Normalmente, este processo é bastante trabalhoso (tratando-se tanto de custo computacional quanto de tempo requerido), uma vez que ele envolve não só o refinamento da melhor solução encontrada até o momento, como também a exploração de soluções completamente novas dentro do espaço de soluções possíveis. Por isso, quando métodos de aprendizado por reforço são aplicados em problemas onde é custoso avaliar-se a qualidade de uma solução encontrada (como, por exemplo, em um problema onde deve ser encontrada a melhor configuração para um braço robótico que deve executar um determinado movimento), é usual o uso de simuladores para, através da simulação do agente em um ambiente virtual, direcionar o processo de aprendizado antes de aplicá-lo no problema real e, deste modo, acelerar o processo de treinamento do agente.

Um simulador, portanto, nada mais é do que uma instância de um problema de aprendfizado por reforço construído em um ambiente virtual, com o objetivo de se acelerar o treinamento de um agente, que, de outra forma, consumiria muito mais tempo. A construção de um simulador normalmente é baseada na união de um motor de física, para simular as condições apresentadas no ambiente, a uma interface gráfica, para a representação visual dos diferentes objetos nele inseridos. Construir um simulador desde a sua

base, entretanto, nem sempre é uma tarefa simples, uma vez que é exigido de quem o desenvolve que este seja versado em ambos os tipos de ferramentas.

Já existem diversos conjuntos de ferramentas (ou *frameworks*) que permitem que sejam construídos simuladores com certo grau de fidelidade e que não possuam uma acentuada curva de aprendizado. Há também, entretanto, um custo associado à adoção de um *framework* para construção de simuladores em um projeto que envolva aprendizado por reforço: este custo refere-se ao tempo necessário para que as ferramentas fornecidas pelo *framework* sejam compreendidas e o cenário proposto seja fielmente reproduzido utilizando-se de todas as funções fornecidas por ele. Ademais, percebe-se que ainda não há uma padronização entre o formato através do qual problemas de aprendizado por reforço são representados por estes simuladores, limitando, desta maneira, a troca de informações entre grupos de pesquisa que usam simuladores diferentes para modelar seus ambientes.

Recentemente, entretanto, foi publicado pela organização OpenAI um conjunto de simuladores chamado Gym, que disponibiliza uma série de modelagens de problemas conhecidos da literatura de inteligência artificial e que também permite que tornem-se públicas as melhores soluções para tais problemas, possibilitando que programadores disponibilizem não só seus algoritmos, como relatórios com informações a respeito do andamento do aprendizado do agente, o que pode ser considerado um enorme passo em direção à padronização da forma com que problemas de aprendizado por reforço e suas respectivas soluções são representados, fornecendo a pesquisadores e programadores uma ferramenta para facilmente comparar um novo algoritmo proposto com outros já amplamente conhecidos e testados.

A documentação do Gym a respeito dos cenários disponibilizados é bastante ampla. Não há, todavia, nenhuma orientação a respeito da construção de cenários que modelem problemas diferentes daqueles disponíveis em seu catálogo. Tais ferramentas seriam úteis na construção de soluções para problemas novos, uma vez que seria possível que soluções para problemas conhecidos poderiam ser facilmente adaptadas aos novos problemas propostos. Este trabalho propõe, portanto, um *framework* para a criação de cenários de aprendizado por reforço que possuam um formato compatível com a API disponibilizada pelo Gym, a fim de facilitar a comparação de um problema novo com soluções que já existam e a reprodutibilidade deste problema, ao mesmo tempo em que seja uma ferramenta de fácil uso e que não exija do programador um extenso estudo da sua documentação.

Neste trabalho, é descrito o processo de criação deste *framework* e é comparada a com-

plexidade de código necessário para criar ambientes de aprendizado por reforço através dele e de outros *framworks* existentes, demonstrando não apenas que os simualdores automaticamente gerados pelo nosso *framework* são compatíveis com APIs padrão na área, mas também que são de fácil especificação, permitindo que a construção de simuladores seja feita através da especificação do problema em uma linguagem descritiva de alto nível e de sintaxe relativamente simples.

Palavras-chave: Inteligência Artificial. Aprendizado por Reforço. Simuladores.

ABSTRACT

This document is an example on how to prepare documents at II/UFRGS using the LaTeX classes provided by the UTUG. At the same time, it may serve as a guide for general-purpose commands. *The text in the abstract should not contain more than 500 words.*

Keywords: Electronic document. LaTeX. ABNT. UFRGS.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

IA Inteligência Artificial

AR Aprendizado por Reforço

SUMÁRIO

1 INTRODUÇÃO

Aprendizado por reforço compreende uma subárea da inteligência artificial que trabalha com a noção de um agente que, inserido dentro de um ambiente, busca uma solução para um determinado problema. Sem nenhuma instrução prévia, é tarefa do agente executar ações dentro do ambiente no qual está inserido, para resolver o problema através delas. A única maneira que o agente tem, entretanto, de avaliar uma ação tomada em um determinado momento, dá-se através da noção de *recompensa*, que nada mais é do que um número escalar que é informado depois de cada ação tomada e que representa a qualidade desta. Com base única e exclusivamente, portanto, nas recompensas recebidas, o agente deve buscar um comportamento (chamado de *política*) que o leve a tomar sempre a melhor ação em um determinado instante, visando maximizar a soma de recompensas recebidas durante uma série de ações. Esta busca pelo melhor comportamento possível é normalmente feita através de uma combinação entre ajustes na melhor política encontrada até o momento (técnica chamada de *exploitation*) e a busca por comportamentos completamente novos ou diferentes do atual (*exploration*).

Diferentemente de outras áreas da IA, como o aprendizado supervisionado, no aprendizado por reforço a qualidade da ação tomada pelo agente não é verificada usando-se como base uma ação ideal ou ótima, conhecida de antemão, e tampouco o agente passa por qualquer tipo de treinamento onde este é exposto a exemplos de ação ótima em cada situação. Tal método de aprendizado é normalmente usado, portanto, em tarefas onde o ambiente é desconhecido pelo agente, e é um modo de aprendizado bastante próximo das maneiras com que animais e humanos buscam formas de exercer tarefas com as quais nunca houve contato prévio.

Um exemplo que pode servir para a compreensão do aprendizado por reforço é o experimento do psicólogo Edward Thorndike [add citation]. No seu experimento, gatos eram colocados em gaiolas fechadas e precisavam encontrar uma maneira de sair para que pudessem consumir uma porção de peixe posicionada próxima à gaiola. Para abri-la, era necessário apenas que uma alavanca presente em seu interior fosse puxada; entretanto, não houve qualquer tipo de instrução prévia: a partir do momento em que os felinos eram trancados nas gaiolas, eles deveriam explorar e, principalmente, interagir de forma autônoma com o interior da gaiola até que, por si mesmos, encontrassem o dispositivo de abertura de seus cárceres. No momento que um gato encontrava a saída, o tempo levado até a sua fuga era anotado e o experimento, repetido. Thorndike percebeu que a partir do

momento em que os gatos aprendiam que era a alavanca o dispositivo responsável pela sua soltura — e que, consequentemente, os permitia que consumissem a porção de peixe —, o tempo transcorrido entre o momento que o animal era recolocado na gaiola e o instante em que ele abria a mesma diminuia consideravelmente. Isso se dá porque, dentre todos os comportamentos adotados dentro da gaiola, o único que era observado pelos gatos como o comportamento que levava à recompensa era o ato de puxar a alavanca. O pesquisador, então, formulou o que ele chamou de "Lei do efeito", que estabelece que comportamentos e ações que, em uma determinada situação, levam a efeitos gratificantes tendem a se repetir e, por sua vez, comportamentos e ações que levem a efeitos indesejáveis ou insatisfatórios tendem a ser abandonados.

Princípios bastante próximos dos postulados pela Lei proposta por Thorndike foram a base para os primeiros experimentos envolvendo aprendizado por reforço, na metade do século passado. Em 1952, um dos grandes expoentes da inteligência artificial, Marvin Minsky, fez um experimento que utilizava uma forma bem simples de AR para simular a maneira com que um rato navegava por um labirinto add citation. No experimento, agentes que simulavam o comportamento de ratos recebiam recompensas mais altas quando desenvolviam um método de busca que achasse a saída do labirinto, e recompensas nulas em caso contrário. Deste então, diversos experimentos foram formulados visando-se a resolução de problemas através de um agente que busca uma solução de maneira praticamente autônoma, sendo guiado apenas pela noção de recompensa, que encapsula o sucesso ou fracasso da solução encontrada.

Aprendizado por reforço é, portanto, um método de aprendizado de máquina pelo qual um agente, ao interagir com um ambiente, executa uma ação e recebe uma recompensa sobre a sua ação tomada, corrigindo seu comportamento de acordo com a recompensa recebida ou de acordo com as modificações produzidas no ambiente por aquela ação, sempre de forma a maximizar o valor esperado relativo às recompensas recebidas pela sua série de decisões. Por causa disto, algoritmos de aprendizado por reforço são amplamente usados em situações onde problemas devem ser resolvidos através de uma sequência de decisões, como quando deseja-se, por exemplo, ensinar um agente a disputar partidas de jogos de tabuleiro (como no xadrez, onde uma série de jogadas leva à vitória) ou ensinar um robô a executar tarefas que exijam uma série de movimentos (como fazer com que um um robô quadrúpede se locomova em trote da maneira mais rápida possível [add citation], por exemplo).

Para facilitar o treinamento de um agente em um problema de aprendizado por re-

força, normalmente emprega-se o uso de simuladores. De matrizes que podem representar um tabuleiro de xadrez até motores de física que representam as leis da física do mundo real, simuladores são usados para modelar problemas de maneira a acelerar o processo de treinamento de um agente, dado que, dentro do ambiente controlado de um simulador, situações que podem demorar algum tempo na vida real (como um braço robótico tentando aprender a melhor forma de pegar um objeto próximo a ele) podem ser retratadas de maneira acelerada. Além do ganho de tempo, elementos de natureza aleatória podem ser facilmente controlados dentro de um ambiente simulado: no problema trabalhado na tese de Andrew Ng add citation, por exemplo, um agente é treinado para ser capaz de controlar um helicóptero, a fim de estabilizá-lo levando em consideração fatores imprevisíveis do sistema (e.g. vento, chuva e demais fatores que podem interferir na estabilidade do veículo). Nota-se que o problema, portanto, tem um objetivo prático: desenvolver um controle para um helicóptero que mantenha sua estabilidade, independentemente de fatores externos. O problema surge na necessidade de treinar o agente: como proceder com o treinamento em ambientes que possuam diferentes níveis de imprevisibilidade e cujos fatores como chuva e vento atuem em diferentes intensidades? Logicamente, o objetivo final do processo é ter um veículo autocontrolado capaz de manter sua estabilidade sob qualquer tipo de interpérie; para que isso seja possível, todavia, é necessário que o agente seja treinado em diferentes tipos de ambiente, o que pode se tornar inviável dadas restrições como o tempo disponível para o projeto e a necessidade dos desenvolvedores de se deslocar até locações onde há condições ideais para o treinamento. A solução que é amplamente usada nesses casos, portanto, é a modelagem de locações em um simulador que representa diferentes tipos de terreno e que permite ao pesquisador que este configure os diferentes fatores do ambiente de acordo com sua necessidade.

Por vezes é necessário, portanto, que haja um ambiente de treinamento que ofereça condições diversas ao agente que está sendo desenvolvido. Esse é um dos cenários que exige a presença de um simulador: um ambiente sob o controle dos desenvolvedores que forneça para eles a liberdade de controlar as condições que serão apresentadas ao agente. Simuladores, deste modo, funcionam como uma ferramenta auxiliar ao processo de desenvolvimento de agentes que atuam no mundo real: inicialmente, treina-se o agente em um ambiente simulado, para depois dar-se prosseguimento ao treinamento em um ambiente real. Mesmo que simuladores não consigam representar com o máximo de precisão as intempéries que podem atingir uma aeronave, por exemplo, eles ainda são úteis nas fases iniciais de treinamento, após as quais o agente está apto a suportar um

ambiente real.

Simuladores também são uma ferramenta que ajuda a mitigar o problema de reprodutibilidade de experimentos. Quando um novo algoritmo de aprendizado por reforço é proposto, por exemplo, há a necessidade de que ele seja facilmente reproduzido por aqueles que têm acesso ao seu artigo de origem. Um simulador, neste caso, pode facilmente recriar as condições em que o algoritmo foi testado, produzindo resultados semelhantes e ajudando na tarefa de verificação do trabalho.

1.1 Motivação

Há diversos *frameworks* capazes de fornecer ao desenvolvedor as ferramentas necessárias para a simulação do seu problema de aprendizado por reforço, mas cada um deles possui as suas próprias limitações: normalmente, o que se observa é uma espécie de compensação entre simplicidade e robustez, onde o usuário do *framework* se vê obrigado a escolher entre uma ferramenta com um alto poder computacional, capaz de simular ambientes com um alto grau de fidelidade, mas que exige uma compreensão maior dos seus mecanismos por parte do desenvolvedor, e uma ferramenta de uso simples, mas que não é tão robusta. Além disso, ainda não há a consolidação de um modelo de representação de simulações na comunidade de programadores e pesquisadores cujo trabalho envolve AR de alguma forma; diferentes ferramentas de construção de simulações, ao adotarem formas diferentes de representarem a maneira com que elementos são simulados e a maneira com que um algoritmo realiza a leitura das informações destes elementos, não permitem que usuários de *frameworks* diferentes troquem informações entre si sem antes realizarem adaptações em seu código.

Recentemente, entretanto, surgiu na comunidade uma ferramenta que fornece simuladores de problemas famosos de aprendizado por reforço e que propõe uma padronização na representação dos problemas: o *framework* chamado Gym, desenvolvido pela OpenAI, uma organização de pesquisadores e entusiastas de inteligência artificial sem fins lucrativos. A ferramenta fornece uma vasta gama de simulações de diversos problemas de aprendizado por reforço e, para cada um deles, há um canal onde pessoas podem submeter suas soluções, sendo montada uma classificação das melhores entre elas, com base em fatores como o tempo necessário para que o agente aprendesse a solucionar o problema. É importante ressaltar, também, que há, para aqueles que submetem suas soluções, a opção de torná-las públicas, permitindo que outras pessoas executem testes com elas.

Através da plataforma proposta pela OpenAI, um grande passo em direção à padronização de simuladores é dado. Através de uma interface pública que serve como ponto intermediário entre o simulador e o algoritmo que tenta resolver o problema, criou-se uma espécie de uniformização não só na maneira como algoritmos realizam a leitura das informações a respeito do ambiente simulado, permitindo que soluções desenvolvidas por pessoas diferentes para um determinado problema tornem-se intercambiáveis, como também na maneira que os resultados de cada solução são representados. Entretanto, apesar da sua vasta documentação a respeito de cada um dos cenários fornecidos, pouco é dito sobre a construção de cenários novos. A proposta do *framework* Gym certamente é bastante inovadora, mas peca no que toca às possibilidades de expansão da ferramenta através de contribuições da comunidade.

O trabalho aqui desenvolvido visa, portanto, fornecer um conjunto de ferramentas que sirva como uma espécie de extensão ao Gym, permitindo que desenvolvedores construam simulações de uma maneira simples, rápida e eficiente, de tal forma que o resultado seja compatível com a API proposta pela OpenAI. Ao propor um *framework* que ao mesmo tempo não exija uma quantidade considerável de tempo para que o seu usuário possa reproduzir o seu problema de AR de maneira fidedigna e que produza resultados que sejam compatíveis com a plataforma Gym, a ferramenta descrita neste trabalho será capaz de fornecer uma ferramenta de uso simples e que garante ao desenvolvedor a possibilidade de trocar informações com a extensa base de usuários do *framework* Gym.

1.2 Estrutura

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

2 CONCEITOS BÁSICOS

Neste capítulo serão introduzidos os conceitos fundamentais para a compreensão do trabalho, através da apresentação da modelagem matemática do problema de aprendizado por reforço através de Processos de Decisão de Markov, bem como a visão geral do fluxo de um algoritmo de AR, com o uso de exemplos quando conveniente. O capítulo também trata do uso de simuladores para a modelagem de tarefas de aprendizado por reforço.

2.1 Definição formal do problema de aprendizado por reforço

O problema de aprendizado por reforço pode ser formalmente definido através de um Processo de Decisão de Markov (também refrido pela sigla MDP, do inglês *Markov Decision Process*). Um MDP é uma ferramenta matemática usada para modelar um processo de tomada de decisão por parte de um agente que, inserido em um ambiente, executa ações que modificam o estado deste ambiente, e cujo resultado imediato de cada ação não é claro para o agente. Usando o conceito de *recompensa*, que é um valor escalar que significa a qualidade de uma ação tomada em um determinado instante, um Processo de Decisão de Markov visa encontrar uma política π que, em um determinado estado, encontre a ação que maximize a soma das recompensas retornadas por uma série de ações. Um MDP pode ser formalmente definido pela tupla (S, A, T, R), onde:

- S é o conjunto de possíveis estados do ambiente;
- A é o conjunto de diferentes ações que podem ser executadas em um determinado estado;
- $T : S \times A \times S \mapsto [0, 1]$: é uma função que dá a probabilidade de o sistema passar para um estado $s' \in S$, dado que o processo estava em um estado $s \in S$ e o agente decidiu executar uma ação $a \in A$ (denotada $T(s'|s, a)$);
- $R : S \times A \mapsto \mathbb{R}$ é uma função que dá a recompensa por uma ação $a \in A$ quando executada no estado $s \in S$.

2.1.1 Estado

Estado, dentro de um contexto de AR, é o conjunto de informações referentes ao agente e ao ambiente em um determinado momento no tempo, e que é utilizado para calcular qual será a próxima ação a ser tomada pelo agente. Considerando que a tomada de ação do agente é regulada por uma função π que mapeia um estado s a uma ação a, o estado, nesse caso, é o subconjunto de todas as informações do agente e do ambiente que serão usadas em s.

A definição desse subconjunto, ou seja, a determinação de quais informações das entidades envolvidas na tarefa de aprendizado irão compor o estado e serão levadas em conta para a próxima tomada de ação do agente, é de responsabilidade do desenvolvedor ou pesquisador que está construindo o agente. Em tarefas simples, como a do exemplo ilustrado pela figura **??**, é trivial a determinação de quais informações farão parte do estado: no exemplo, é óbvio que a posição da área em azul e das partes do braço mecânico serão levadas em conta. Em outros problemas, entretanto, onde há agentes mais complexos lidando com múltiplos objetos no ambiente, é menos óbvia a tarefa de determinar o conjunto s que será fornecido como entrada à função π. Isto pode influenciar no tempo necessário para a conclusão de um projeto que envolva aprendizado por reforço, uma vez que vários conjuntos diferentes de informações devem ser testados de maneira iterativa até que seja encontrado um estado que minimize o tempo até a conclusão da tarefa e torne o agente mais capaz de resolver o problema.

Estado, dentro de um contexto de aprendizado por reforço, portanto, é um subconjunto contido no conjunto de todas as possíveis informações oriundas de todos os elementos envolvidos dentro de uma tarefa de aprendizado — incluindo informações sobre o ambiente e sobre o próprio agente) — e que é usado como entrada na função do agente que mapeia um estado s a uma ação a. As informações que compõem o estado, entretanto, são fruto de uma decisão humana, por parte de quem está desenvolvendo o projeto, e, considerando que decisões diferentes podem trazer resultados diferentes (dependendo da complexidade do projeto), pode ser necessário que diferentes formas de compor o estado sejam testadas a fim de que se encontre qual delas é a mais satisfatória para o problema a ser resolvido.

2.2 Aprendizado por Reforço

Aprendizado por reforço compreende uma subárea da inteligência artificial que trabalha com a noção de um agente que explora um ambiente a fim de buscar uma solução (comportamento) para determinado problema. Sem nenhum instrução prévia, é tarefa do agente buscar, dentro do espaço de soluções possíveis, uma maneira satisfatória de resolver o problema através da sua interação com o ambiente. Para determinar se uma solução encontrada é satisfatória, usa-se, em aprendizado por reforço, a noção de recompensa. A recompensa é um sinal númerico e pode ser calculada após cada ação tomada pelo agente, ou ao final de cada episódio (*delayed reward*), é um número que encapsula a qualidade daquela ação ou do conjunto de várias ações tomadas ao longo do tempo. Com base única e exclusivamente, então, nas recompensas recebidas pelo agente, o agente deve buscar uma solução (comportamento) que a maximize (de forma a resolvar o problema). Esta busca é normalmente feita através de uma combinação entre ajustes nas ações que o agente tomou e que levaram à maior recompensa até o momento (*exploitation*) e a avaliação do resultado de ações completamente novas ou desconhecidas pelo agente (*exploration*).

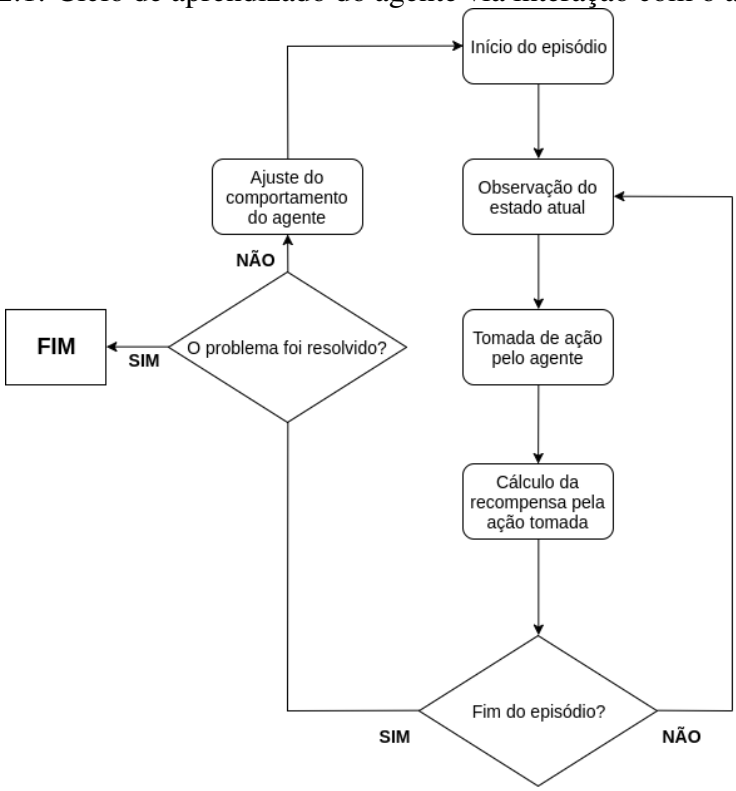
Diferentemente de outras áreas da IA, como o aprendizado supervisionado, no aprendizado por reforço a qualidade da ação tomada pelo agente não é verificada usando-se como base uma ação ideal ou ótima, conhecida de antemão, e tampouco o agente passa por qualquer tipo de treinamento onde este é exposto a exemplos de ação ótima em cada situação. Tal método de aprendizado é normalmente usado, portanto, em tarefas onde o ambiente é desconhecido pelo agente, e é um modo de aprendizado bastante próximo das maneiras com que animais e humanos buscam formas de exercer tarefas com as quais nunca houve contato prévio.

Um exemplo que pode servir para a compreensão do aprendizado por reforço é o experimento do psicólogo Edward Thorndike. No seu experimento, gatos eram colocados em gaiolas fechadas e precisavam encontrar uma maneira de sair para que pudessem consumir uma porção de peixe posicionada próxima à gaiola. Para abri-la, era necessário apenas que uma alavanca presente em seu interior fosse puxada; entretanto, não houve qualquer tipo de instrução prévia: a partir do momento em que os felinos eram trancados nas gaiolas, eles deveriam explorar e, principalmente, interagir de forma autônoma com o interior da gaiola até que, por si mesmos, encontrassem o dispositivo de abertura de seus cárceres. No momento que um gato encontrava a saída, o tempo levado até a sua fuga

era anotado e o experimento era repetido. Thorndike percebeu que, uma vez que os gatos aprendiam que era a alavanca o dispositivo responsável pela sua soltura – e que, consequentemente, os permitia que consumissem a porção de peixe –, o tempo transcorrido entre o momento que o animal era recolocado na gaiola e o instante em que ele abria a mesma diminuia consideravelmente. Isso se dá porque, dentre todos os comportamentos adotados dentro da gaiola, o único que era observado pelos gatos como o comportamento que levava à recompensa era o ato de puxar a alavanca. O pesquisador, então, formulou o que ele chamou de "Lei do efeito", que estabelece que comportamentos e ações que, em uma determinada situação, levam a efeitos gratificantes tendem a se repetir e, por sua vez, comportamentos e ações que levem a efeitos indesejáveis ou insatisfatórios tendem a ser abandonados.

Princípios bastante próximos dos postulados pela Lei proposta por Thorndike foram a base para os primeiros experimentos envolvendo aprendizado por reforço, na metade do século passado. Em 1952, um dos grandes expoentes da inteligência artificial, Marvin Minsky, fez um experimento que utilizava uma forma bem simples de AR para simular a maneira com que um rato navegava por um labirinto add citation. No experimento, agentes que simulavam o comportamento de ratos recebiam recompensas mais altas quando desenvolviam um método de busca que achasse a saída do labirinto, e recompensas baixas em caso contrário. Deste então, diversos experimentos foram formulados visando-se a resolução de problemas através de um agente que busca uma solução de maneira praticamente autônoma, sendo guiado apenas pela noção de recompensa, que encapsula o sucesso ou fracasso da solução encontrada. Aprendizado por reforço é, portanto, um método de aprendizado de máquina pelo qual um agente, ao interagir com um ambiente, executa uma ação e recebe uma recompensa sobre a sua ação tomada, corrigindo seu comportamento de acordo com a recompensa recebida, sempre de forma a maximizá-la. A figura 2.1 ilustra o fluxo que geralmente ocorre em tarefas de aprendizado por reforço. Antes de formalizarmos matematicamente o problema e apresentarmos um algoritmo clássico na seção [ver secao nova que eu sugeri criar, aonde tu vai falar de processos de markov e do Q-Learning], iremos discutir nas subseções seguintes, de forma intuitiva, os conceitos mais importantes relacionados a problemas de aprendizado por reforço: o conceito de agente, ambiente, estado, ação, recompensas e episódios. Posteriomente, esses conceitos serão formalizados matematicamente, e então iremos discutir, na seção [Simuladores de AR], como simuladores podem ser usados para acelerar o processo de treinamento de agentes via diferentes algoritmos de aprendizado.

Figura 2.1: Ciclo de aprendizado do agente via interação com o ambiente



2.2.1 Agente

Agente é a entidade dotada de capacidade de aprendizado que tenta extrair do ambiente a melhor maneira — de acordo com uma métrica de performance, pré-estabelecida implicitamente via uma função de recompensa — de executar uma determinada tarefa. No exemplo do gato de Thorndike, o agente é o próprio gato e as ações que ele executa são as possíveis interações com o interior da gaiola, a fim de abri-la. É o agente, portanto, a entidade responsável por decidir, através de uma observação do estado do ambiente, qual ação será tomada, visando sempre obter uma recompensa mais alta do que a maior recompensa recebida até o presente momento.

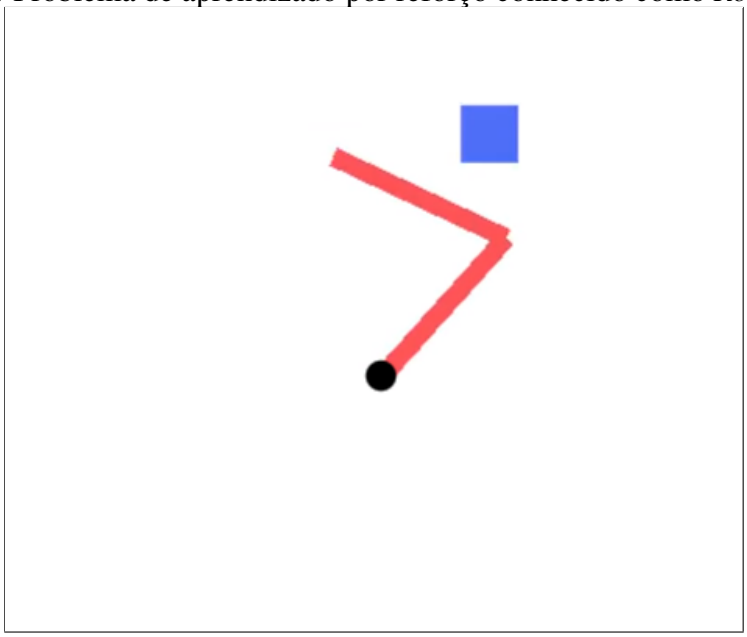
Em alguns problemas, pode ser interessante que o estado do próprio agente seja lido antes que a próxima ação a ser tomada seja escolhida. Ou seja, pode ser relevante, a fim de determinar a melhor ação do agente, não apenas observar características do estado externo ao agente, tais como a posição de diferentes obstáculos, mas também a observação de características do estado interno do agente, tal como seu nível atual de bateria e o posicionamento físico das suas diferentes partes. Nesse caso, informações são coletas sobre todas as partes que formam o agente, tal como, por exemplo, a posição espacial de uma de suas pernas. No exemplo descrito na seção 2.2.2, é imprescindível que o

agente leve em consideração a posição do braço mecânico para decidir qual movimento será executado em seguida. Isso implica que a especificação do agente corresponde não apenas à descrição de seus componentes físicos, como no caso de um robô, mas também na definição de quais informações irão compor o estado com base no qual tal agente irá escolher ações; em particular, parte do estado pode envolver a observação de sensores internos ao próprio agente. Mais detalhes sobre isso serão discutidos na Secao [secao onde fala de Estado].

2.2.2 Ambiente

O ambiente é o espaço onde o agente está inserido e de onde ele extrai as informações necessárias para decidir que ação será tomada. No ambiente podem se encontrar, por exemplo, objetos com os quais o agente deverá interagir para que o problema seja resolvido. Em um famoso problema de AR dentro do campo da robótica, um agente que simula um braço mecânico deve mover-se de tal maneira que sua extremidade esteja dentro de uma área demarcada no ambiente, área esta que é colocada em posições aleatórias ao início de cada episódio de treinamento (figura 2.2).

Figura 2.2: Problema de aprendizado por reforço conhecido como *Robotic Arm*.



Através de uma leitura da posição da área demarcada em azul, o agente deve decidir como irá ser executado o movimento do braço mecânico, composto de duas peças unidas, uma delas afixada no ponto de cor preta, no centro da tela, de modo a tocar a

área azul com a extremidade do braço, fazendo sempre o mínimo possível de esforço, de forma a maximizar a recompensa (neste exemplo, a recompensa também toma como base a energia desprendida pelo braço mecânico para tocar a área demarcada). O ambiente compreende, por conseguinte, não só o espaço físico (real ou simulado) onde o agente está inserido e onde ele executa as suas ações; o ambiente é, além disso, a fonte de onde ele extrai as informações necessárias para decidir o que fazer, e também contém a especificação da tarefa específica que o agente irá ter que resolver—e.g., mover o braço para uma posição específica, ou então empurrar um determinado objeto, etc. certeza que, filosoficamente falando, não é o agente que carrega a especificação do problema? na real, a especificação do problema é tratada no meu tcc como algo alheio ao agente e ao ambiente, ou seja, não pertencente a nenhum dos dois. acho que só serviria pra confundir ao invés de explicar que o ambiente de alguma maneira "carrega"consigo a definição do problema. (tanto é que no meu trabalho a função que lê o estado e define se o problema já foi resolvido é separada dos dois)

2.2.3 Estado

Estado, dentro de um contexto de AR, é o conjunto de informações referentes ao agente e ao ambiente em um determinado momento no tempo, e que é utilizado para calcular qual será a próxima ação a ser tomada pelo agente. Considerando que a tomada de ação do agente é regulada por uma função π que mapeia um estado s a uma ação a, o estado, nesse caso, é o subconjunto de todas as informações do agente e do ambiente que serão usadas em s.

A definição desse subconjunto, ou seja, a determinação de quais informações das entidades envolvidas na tarefa de aprendizado irão compor o estado e serão levadas em conta para a próxima tomada de ação do agente, é de responsabilidade do desenvolvedor ou pesquisador que está construindo o agente. Em tarefas simples, como a do exemplo ilustrado pela figura **??**, é trivial a determinação de quais informações farão parte do estado: no exemplo, é óbvio que a posição da área em azul e das partes do braço mecânico serão levadas em conta. Em outros problemas, entretanto, onde há agentes mais complexos lidando com múltiplos objetos no ambiente, é menos óbvia a tarefa de determinar o conjunto s que será fornecido como entrada à função π. Isto pode influenciar no tempo necessário para a conclusão de um projeto que envolva aprendizado por reforço, uma vez que vários conjuntos diferentes de informações devem ser testados de maneira iterativa

até que seja encontrado um estado que minimize o tempo até a conclusão da tarefa e torne o agente mais capaz de resolver o problema.

Estado, dentro de um contexto de aprendizado por reforço, portanto, é um subconjunto contido no conjunto de todas as possíveis informações oriundas de todos os elementos envolvidos dentro de uma tarefa de aprendizado — incluindo informações sobre o ambiente e sobre o próprio agente) — e que é usado como entrada na função do agente que mapeia um estado s a uma ação a. As informações que compõem o estado, entretanto, são fruto de uma decisão humana, por parte de quem está desenvolvendo o projeto, e, considerando que decisões diferentes podem trazer resultados diferentes (dependendo da complexidade do projeto), pode ser necessário que diferentes formas de compor o estado sejam testadas a fim de que se encontre qual delas é a mais satisfatória para o problema a ser resolvido.

2.2.4 Ação

Um agente de aprendizado por reforço deve, a cada momento tomar uma ação. Através da sua ação, o agente interage com o ambiente (e consigo mesmo) para provocar uma mudança no estado do sistema (e, conforme mencionado anteriormente, por estado do sistema compreende-se o estado tanto do agente quanto do ambiente a fim de resolver o problema. Esta concepção de ação como uma interação com o ambiente que resulta em uma mudança no estado do sistema é inerente ao aprendizado por reforço: a grande diferença entre os diferentes algoritmos de AR, entretanto, reside em como a próxima ação é escolhida e em como o agente atualiza o seu comportamento, baseado na observações sobre os resultados (e.g. recompensa recebida) após a execução da ação escolhida, para atualizar seu comportamento.

2.2.5 Episódio

Um episódio é, basicamente, uma fase de treinamento de um agente de aprendizado por reforço. A fase inicia com a criação do ambiente e a inserção do agente nele (inicialização) e, após isso, o agente interage com o ambiente até que o estado do sistema seja um estado final. Em um problema que, por sua complexidade, vem sendo usado como exemplo no uso de redes neurais profundas em treinamentos por aprendizado por

Figura 2.3: Fim de um episódio: o agente não foi capaz de superar um obstáculo.

reforço, um agente humanóide precisa aprender a caminhar: um episódio termina, portanto, quando o agente cai no chão, quando não consegue superar algum obstáculo (figura 2.3) ou quando um determinado número de iterações é atingido.

2.2.6 Recompensa

Recompensa é um número atribuído a uma ação do agente que encapsula o sucesso desta quando o agente se encontra em um determinado estado. Para ações que sejam avaliadas como melhores, a recompensa é mais alta; para ações que devem ser tratadas como indesejáveis pelo agente, a recompensa atribuída é mais baixa e, portanto, o agente deve corrigir-se a fim de que a ação que levou à baixa recompensa seja refinada ou abandonada de vez. É importante notar que nem sempre uma ação é diretamente ligada a uma recompensa: no caso de *delayed reward*, a recompensa não é entregue ao agente no exato momento em que ele toma a decisão, sendo apenas revelada ao agente ao fim de uma sequência de ações. No caso de um agente que aprende a jogar damas, por exemplo, a recompensa é somente atribuída ao final da partida.

Isto leva a um problema bastante estudado no ramo do aprendizado por reforço chamado "atribuição de crédito": em uma série de ações que levaram a uma recompensa baixa, como encontrar, dentre todas as ações tomadas, apenas aquelas que foram responsáveis pelo baixo desempenho? - eu nao sei onde tava querendo chegar quando escrevi isso e vou remover. azar

A recompensa é, portanto, um número que encapsula o quão adequada foi uma ação ou série de ações tomadas pelo agente para a resolução do problema. Para ações que tornam o agente mais próximo de resolvê-lo, é ideal que sejam recebidas recompensas mais altas. Para ações tidas como indesejáveis ou que não mudem o estado do sistema

para um mais próximo do estado final, por outro lado, devem resultar em recompensas mais baixas para que o agente corrija seu comportamento.

2.3 Definição formal de um problema de aprendizado por reforço

O problema do aprendizado por reforço pode ser formalizado como um Processo de Decisão de Markov. Um Processo de Decisão de Markov representa um problema de aprendizado por reforço como uma constante troca de estados, onde cada troca representa uma ação e é seguida de uma recompensa. Cada estado, nesse caso, é uma observação do agente, e a transição é uma ação tomada por ele, que modifica o ambiente e leva a uma observação (ou seja, a outro estado) diferente. Um PDM modela uma função que procura maximizar o total de recompensas recebidas ao longo do tempo. A função é representada da seguinte maneira:

$$\max \sum_{t=0}^{t=\infty} \gamma^t r(s(t), a(t)) \tag{2.1}$$

Na fórmula, $r(s(t), a(t))$ é uma função que associa, em um instante de tempo t uma recompensa r a uma ação a tomada quando o estado observado é s. O objetivo da equação, portanto, é maximizar as recompensas recebidas ao longo do tempo tomando em cada momento a ação que leva à melhor recompensa. O coeficiente γ, neste caso, é um número real no intervalo $(0, 1]$ que determina a diferença de importância entre recompensas recebidas no presente e no futuro.

A fórmula 2.1 capta de um modo geral o objetivo de um algoritmo de aprendizado por reforço: escolher, em um determinado instante de tempo, a ação que levará à maior recompensa. Porém, ainda falta definir como será escolhida a melhor ação a. Para isto, é necessário definir uma *política*. Uma política pode ser representada como uma função $\pi(s) \rightarrow a$ que, recebendo um estado s, define que uma ação a deve ser tomada. A fim de que, conforme definido na fórmula 2.1, sempre seja escolhida a melhor ação (i.e. a ação que leva a melhor recompensa), usa-se uma *política ótima* $\pi* : S \rightarrow A$, que recebe um estado $s \in S$ e retorna uma ação ótima $a \in A$.

Mesmo definindo matematicamente o problema de aprendizado por reforço em 2.1 e definindo o que é uma política ótima, resta saber como encontrar uma política ótima. Em inteligência artificial, a noção de *aprendizado* trata de uma correção que o agente executa em si mesmo, levando em consideração seu erro e um coeficiente de aprendizado, que determina o impacto que o último erro terá na atualização do agente em um determi-

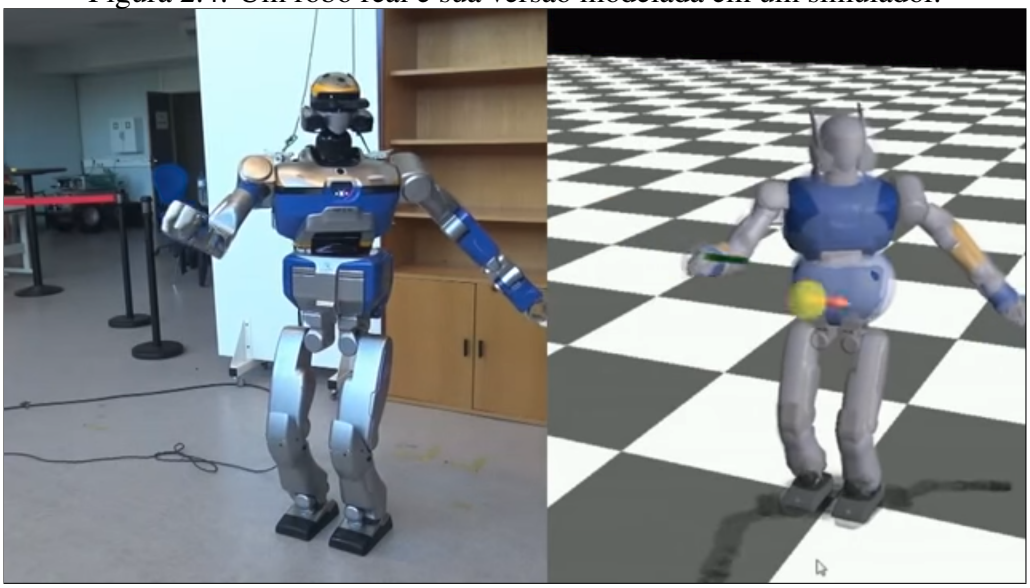
nado momento. Há muitos algoritmos de aprendizado por reforço que funcionam dessa maneira, e um dos mais estudados deles é chamado Q-learning.

2.4 Q-learning

Coloco depois. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

2.5 Simuladores de aprendizado por reforço

Figura 2.4: Um robô real e sua versão modelada em um simulador.



Aprendizado por reforço lida com problemas onde um agente, inserido em um ambiente, tenta resolver um determinado problema através de uma série de ações. Em alguns problemas, é de interesse do responsável pelo desenvolvimento do agente que se

possa modelar casos nos quais o agente está inserido em um ambiente estocástico (i.e. que apresenta fatores aleatórios que influenciam em seu comportamento). No problema trabalhado na tese de Andrew Ng add citation, por exemplo, um agente é treinado para ser capaz de controlar um helicóptero, a fim de estabilizá-lo levando em consideração fatores imprevisíveis do sistema (e.g. vento, chuva e demais fatores que podem interferir na estabilidade do veículo). Nota-se que o problema, portanto, tem um objetivo prático: desenvolver um controle para um helicóptero que mantenha sua estabilidade, independentemente de fatores externos. O problema surge na necessidade de treinar o agente: como proceder com o treinamento em ambientes que possuam diferentes níveis de imprevisibilidade e cujos fatores como chuva e vento atuem em diferentes intensidades? Logicamente, o objetivo final do processo é ter um veículo autocontrolado capaz de manter sua estabilidade sob qualquer tipo de interpérie; para que isso seja possível, todavia, é necessário que o agente seja treinado em diferentes tipos de ambiente, o que pode se tornar inviável dadas restrições como o tempo disponível para o projeto e a necessidade dos desenvolvedores de se deslocar até locações onde há ambientes ideais para o treinamento. A solução que é amplamente usada nesses casos, portanto, é a modelagem de locações em um simulador que representa diferentes tipos de terreno e que permite ao pesquisador que este configure os diferentes fatores do ambiente de acordo com sua necessidade.

Além da economia de tempo e da liberdade para modelar de forma controlada diferentes tipos de ambiente com comportamento estocástico, simuladores também oferecem uma maneira de analisar o desempenho de diferentes algoritmos de aprendizado quando aplicados ao mesmo problema, ou o contrário, que ocorre quando um mesmo algoritmo de aprendizado é aplicado a problemas diferentes a fim de que possa ser avaliada a sua generalidade. Esse foi um dos objetivos do desenvolvimento da plataforma Gym, da OpenAI, uma organização sem fins lucrativos que desenvolve projetos com foco em inteligência artificial. Com um catálogo de diversos cenários de aprendizado por reforço (chamados de *environments*) oriundos da literatura de IA e até mesmo de jogos feitos para antigas plataformas de 8 *bits*, esta plataforma oferece uma API (Interface Pública de Aplicação, em tradução livre; espécie de ponto de acesso ao qual programas podem se conectar, se seguirem um determinado formato) que permite que qualquer pessoa desenvolva seu algoritmo de AR para um determinado cenário e submeta-o a uma espécie de *ranking*, que elenca os responsáveis pelos algoritmos que resolveram o problema de maneira mais eficiente, levando em consideração a maior recompensa recebida e o tempo levado para resolver o problema. Sua desvantagem, entretanto, é que o catálogo de *environments* oferecido pelo

Gym é fixo, ao mesmo tempo em que não existe nenhum *framework* capaz de oferecer ao desenvolvedor as ferramentas necessárias para produzir cenários que são compativeis com a API do Gym. Isso possibilitaria, por exemplo, uma busca rápida de algoritmos que comprovadamente funcionam em cenários conhecidos a fim de que estes sejam mudados para sua aplicação em ambientes novos. Este trabalho propõe um *framework* que auxilia na criação de tais cenários, sem que o programador tenha que lidar com questões de programação de baixo nível como a troca de informações entre um motor de física, responsável pelas simulações, e uma biblioteca gráfica, que transforma o ambiente simulado em algo que pode ser representado na tela através de *pixels*.

aqui eu ainda tenho que alterar, com base no que vou colocar no capítulo de estado da arteNo próximo capítulo, será feito um breve estudo dos simuladores Gym e MuJoCo secao de traçando suas principais vantagens e deficiências, através de uma análise comparativa que relacionará seus pontos fortes e desvantagens com o que é oferecido pelo simulador descrito no capítulo 4 deste trabalho. gym e mujoco sao comparaveis? gym é uma API, mujoco é um gerador de codigo. Tem que falar do Gym no inicio, clarificando que nao se trata de um simulador nem de um gerador de ambientes/simualdores, e sim um padrao/API que, se um deterinado ambiente/simulador seguir, ele pode ser usado para avaliar qualquer um dos N algoritmos de aprendizado que ja foram implementados e que seguem essa API. Aí precisa falar do mujoco e de outros geradores de codigo de ambientes/simuladores. Eu tinha mandado, num email, alguns links. Tem o framework dos italianos, e tinha um outro negocio que eu tinha mandado por facebook que era de um negocio meio em modo-texto. Nao lembro qual era. Mas tem mais coisa que só o mujoco; tem que achar os links que eu tinha mandado por email.

2.6 A linguagem YAML

Coloquei essa parte aqui porque faço referência a ela várias vezes durante o desenvolvimento. YAML é uma linguagem de marcação utilizada para serialização de dados. Ela é usada normalmente para arquivos de configurações de sistemas (como é o caso do *framework*) Ruby on Rails, que permite a criação de servidores *web* cujo acesso a seu banco de dados, por exemplo, é determinado por configurações salvas em um arquivo neste formato. Apesar de ter sido concebida como uma linguagem de marcação, pelo fato de sua sigla originalmente significar *Yet Another Markup Language* ("mais uma linguagem de marcação", em tradução livre), mais tarde sua sigla se tornou recursiva e seu

significado mudou para *YAML: YAML Ain't Markup Language* ("YAML não é linguagem de marcação"), como forma de reposicionamento por parte seus desenvolvedores, que buscavam redirecionar o uso da linguagem para representações de dados, uso este que é bem diverso de marcação de texto.

Sua sintaxe permite a declaração de vetores associativos (também conhecidos como "dicionários"), onde valores são associados a chaves, sendo estas normalmente de caracteres. A possibilidade de definir dicionários, juntamente com a capacidade da linguagem de estruturar dados sob a forma de listas, permite que o programador defina estruturas bastante complexas, mas que são sintaticamente mais simples do que se os mesmos dados fossem definidos usando-se de outras linguagens de marcação, como XML e JSON.

Um exemplo do uso da linguagem YAML é dado abaixo. Nele, define-se uma lista composta de dois funcionários que possuem informações como nome, posição e uma lista de habilidades.

```
  # Employee records
-  martin:
    name: Martin D'vloper
    job: Developer
    skills:
      - python
      - perl
      - pascal
-  tabitha:
    name: Tabitha Bitumen
    job: Developer
    skills:
      - lisp
      - fortran
      - erlang
```

A ferramenta de modelagem de cenários de aprendizado por reforço descrita neste trabalho oferece duas maneiras de modelagem de entidades de AR e uma delas funciona através da leitura de um *script* YAML que descreve, de maneira hierárquica, as entidades e suas propriedades. Uma descrição detalhada de como descrever um problema de AR

através de um arquivo YAML é dada no capítulo 5 deste documento. er um problema de AR através de um arquivo YAML é dada no capítulo 5 deste documento.

3 ESTADO DA ARTE

3.1 Gym

Gym é um simulador desenvolvido pela OpenAI nao é um simulador. É uma API, uma empresa sem fins lucrativos que desenvolve pesquisas em inteligência artificial com o intuito de "desenvolver IA amigável de tal forma que a humanidade como um todo beneficie-se dela". Seu trabalho com o Gym, portanto, não tem por objetivo somente prover a pesquisadores do mundo todo uma ferramenta com a qual seja possível desenvolver, avaliar e comparar diferentes algoritmos de aprendizado por reforço, mas também fornecer a todos que usam o simulador um canal por onde se possa compartilhar ideias e comparar métodos. O Gym nasceu, portanto, da necessidade de se ter uma ferramenta capaz de fazer o *benchmark* de algoritmos de AR. Enquanto em outras áreas da IA, como aprendizado supervisionado, já existe grandes bases de dados como a ImageNet, com centenas de milhares de imagens já etiquetadas para uso principalmente em projetos de IA que envolvam reconhecimento de imagens, não existia, até o momento em que o Gym foi concebido, uma biblioteca de ambientes e resultados que fosse grande o suficiente e que, principalmente, fosse fácil de usar tem que clarificar que é uma API que tu usa pra especificar tanto os ambientes de RL quanto os algoritmos de aprendizado, e que como toda a interface fica padronizada, é possivel fazer combinacoes e testar qualquer algoritmo em qualquer ambiente, e é possível reproduzir resultados de outros algoritmos, facilmente comparar novos algoritmos com aqueles existentes, etc. Aí precisa dizer concretamente o que o Gym faz: qual a interface de métodos principais que a pessoa precisa implementar pra especificar um ambiente, e quais precisa implementar pra especificar um agente de aprendizado. Diz aí tipo 'o metodo takeAction, p.ex., implementa a politica π de escolha de acoes. O metodo runAction (to inventendo os nomes, nao lembro de cor) pega uma acao e o estado atual, e devolve o proximo estado do ambiente e a recompensa; ou seja, ele corresponde à implementacao computacional da funcao de transicao T do MDP e da funcao de recompensa R'. Acho que falta, nessa parte, falar de forma mais concreta o que que é o que o Gym faz/especifica, pra conseguir fazer essa padronizacao. Isso é importante porque todo o teu argumento, depois, é que o teu framework, ao gerar codigo compativel com o Gym, permite a geracao facilitada de novos ambientes para teste, avaliacao e comparacao de algoritmos de RL. Quando for explicar o Gym, tu poderia nao só colocar aquelas imagens, mas realmente descrever alguns deles, como exemplos.

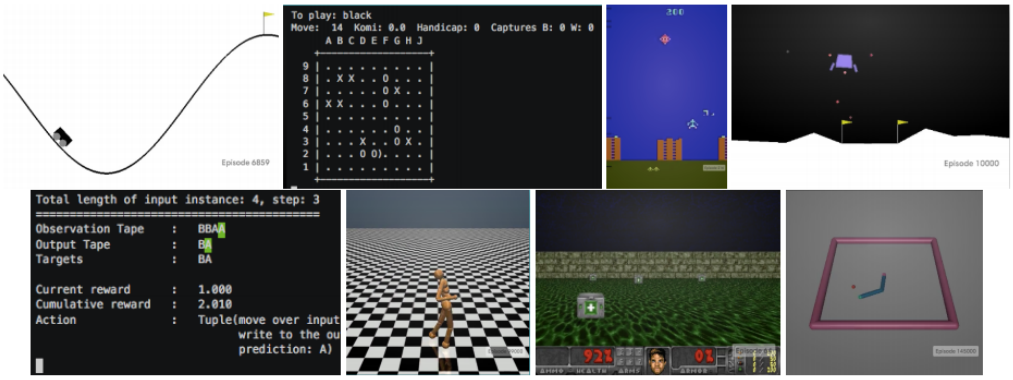
Tipo 'Alguns exemplos de ambientes clássicos usados como benchmark de algoritmos de aprendizado por reforco, tais como poll balancing e mountain car, por exemplo, estao disponiveis na plataforma Gym. O mountain car corresponde ao problema de (...). Para que um desenvolvedor com interesse em implementar e validar um novo algoritmo de RL pudesse testar sua criacao no mountain car, bastaria utilizar o seguinte codigo Python: (...). Caso quisesse testar esse metodo codigo no problema de poll balancing, bastaria alterar o codigo do seu agente de aprendizado da seguinte forma: (...). Como pode ser visto, uma vez que ambientes e agentes sao implementados seguindo a API do Gym, torna-se trivial instanciar comparacoes de qualquer método em qualquer tarefa de aprendizado'. Aí fala 'alem de ambientes classicos, cenarios mais recentesmente propostos como benchmark, tais como aqueles envolvendo jogos de atari, tambem estao disponiveis'. Aí fala um pouco deles. Depois de falar sobre a parte de cenarios/ambientes que tem no Gym, fala sobre como é sao disponibilizados os algoritmos de aprendizado que existem nele. As pessoas enviam codigo pra resolver um ambiente especifico, e o codigo é rodado la? Elas rodam localmente e loggam os resultados, e só os resultados sao enviados, com alguma maneira de eles verificarem que nao foram adulterados? Tem alguma maneira facil de baixar os top N best-ranked algorithms pra um detemrinado problema? Coisas desse tipo. Aí descreve como ter acesso aos melhores algoritmos submetidos pra um dado ambiente é util pra que depois tu possa pegar o algoritmo que tu inventou e comparar com o estado-da-arte, de acordo com as pessoas que submetem pro Gym; fala que esse ranking é interessante porque consiste em resultados que podem ser reproduzidos, e nao apenas em resultados que *apenas* foram publicados em artigos cientificos, mas para os quais frequenemtenete nao se tem acesso ao fonte, e/ou nao se conhece como os autores ajustaram os meta-parametros tipo taxa de aprendizado, etc. Ou seja, facilidade reproducibility e permite com que tu rapidamente avalie o teu algoritmo contra varios existentes, ou entao teste algoritmos existentes (publicados la) em um novo tipo de ambiente/tarefa de aprendizado, no qual tu possa ter interesse.

Os cenários disponibilizados pelo Gym são bastante diversos e vão desde problemas em modo texto até jogos de plataformas bidimensionais (figura 3.1). Para cada um dos *enviroments* disponíveis, há, na página do projeto, um *ranking* dos algoritmos que resolveram o problema da maneira mais eficiente, levando dois fatores em consideração: maior recompensa recebida e quantidade de episódios necessários para que o agente finalmente resolvesse o problema proposto. Todavia, o Gym ainda carece de uma extensão que permita que pesquisadores compartilhem não só suas soluções, mas também ambi-

entes totalmente novos ou adaptações de ambientes que já existem o que tu quer dizer com isso é que o pacote, como tu baixa da pagina, vem só com um conjunto pre-aceito de ambientes, ne? Que as pessoas tem como submeter seus agentes de aprendizado, mas nao seus ambientes? Isso é uma limitacao importante? Nao poder compartilhar os ambientes, digo? Porque caso tu enfatize essa limitacao, a expectativa do leitor seria que tu iria resolver ela (uma plataforma pra poder submeter novos ambinetes). Acho que a tua contribuicao nao é essa, e sim resolver uma outra limitacao dele: que ele especifica só a API, mas aí se tu quer criar um ambiente novo do zero, que siga a API deles, tu precisa ir aprender uns motor de fisica do zero, e tambem precisa aprender a API da plataforma e tudo mais. O que tu vai fazer é propor um framework pra geracao automatica de codigo que pegue a especificacao de alto-nivel de um ambiente e compile ela pra codigo de mais baixo nivel que implementa essa descricao em um motor de fisica, e de forma que tudo seja compativel com Gym, ou seja, de forma que instantaneamnete dê pra disponbilizar pra toda comunidade esse novo ambiente, e que se instantanemanete tenha acesso/possibilidade de avaliar N algoritmos existentes no ambiente que tu acabou de inventar.

O *Barbell*, desenvolvido e descrito neste trabalho, é uma tentativa de resolver, mesmo que em partes, este problema: ao fornecer ao desenvolvedor uma maneira de criar seus próprios ambientes, cria-se a possibilidade de expandir a biblioteca do Gym; por outro lado, ao desenvolver cenários que de certa forma recordem outros que já existem, o *Barbell* permite que o desenvolvedor parta de algoritmos que já foram desenvolvidos ao invés de se encontrar obrigado a construir sua solução do zero. O método de construção de cenários e a maneira como os resultados são apresentados, em comparação com os resultados fornecidos pelos cenários padrão do Gym, será descrito no próximo capítulo.

Figura 3.1: Alguns dos cenários disponíveis na base do Gym.



3.2 MuJoCo

IMPORTANTE: que outros artigos discutem frameworks com esse mesmo objetivo? DISCUTIR ISSO NOS TRABALHOS RELACIONADOS:

- rllab, moveit <https://stackoverflow.com/questions/48490828/should-i-using-mujoco-or-moveit-for-r
- quais outros motores de fisica as pessoas usam? gazebosim.org, vrep;
- como as pessoas implementam os dominios compativeis com o RLGlue?
- que outros simuladores de robotica sao utilizados? p.ex. <https://en.wikipedia.org/wiki/Robotics_simulator>;
- andersonrochatavares@gmail.com, perguntar se ele conhece artigos e/ou frameworks que se costuma usar para gerar jogos/dominios;

ver comentario no inicio do capto. Acho que o gym e o mujoco nao sao comparaveis, porque Gym=API, e mujoco=framework pra geracao de ambientes/simulacoes. ALgo comparavel com mujoco seriam aqueles outros trabalhos que eu mencionei em algum dos emails anteriores

4 DESENVOLVIMENTO???

Realmente não sei o que colocar aqui. Acho que tenho que convencer nessa parte de que meu trabalho é muito mais foda e muito mais intuitivo do que todos os outros trabalhos que já foram desenvolvidos até hoje, certo? Bruno, por favor, me ajuda. Aqui precisa descrever algo como o que eu tinha mencionado numa thread de emails com subject 'TODO + exemplo de TCC', e também ao longo da thread com subject 'TCC. estou desesperado'. Fala do que tu fez, qual a sintaxe da linguagem de alto nivel que tu criou, o que tu consegue especificar com ela, como tu especifica as propriedades fisicas do agente, as acoes que ele pode executar, a funcao de recompensa, etc. Fala como tu faz pra pegar esse negocio e compilar pra codigo de mais baixo nivel usando um motor de fisica. Diz como tu escolheu essa sintaxe especifica, ou seja, porque tu nao fez uma coisa mais geral (mas mais dificil de usar), tipo mujoco. Pra fazer isso tu pode fazer um link com o que falou antes, na secao do mujoco, onde tu fala que ele é super generico mas é de dificil uso pra leigos, que tem uma curva acentuada de aprendizado, etc, e que aqui tu foca em poder criar ambientes que tenham essa estrutura de um jogo, que é o que acontece na imensa maioria dos ambientes existentes em benchmarks padroes da area e dentre os ambientes no Gym. Fala de como tu organizou o teu codigo, quais metodos da API do Gym ele implementa automaticamente, quais ele implementa parcialmente mas deixa o usuario completar com codigo mais refinado (tipo a definicao de novas acoes, ou a definicao da funcao de recompensa). A gente tinha conversado tambem sobre aquela questao de haver N sensores no ambiente/agente, e aí ter uma funcao que calcula o estado, que vai ser um subset desses sensores (e/ou novos valores calculados com base neles), e que vai ser efetivamente o que vai ser usado para escolher acoes, etc. Fala como isso funciona. Esse tipo de coisa

Neste capítulo, será mostrado o funcionamento dos mecanismos fornecidos pelo Barbell para a modelagem de ambientes e agentes que nele operam, bem como uma visão geral sobre a sua estrutura básica, com uma breve explicação das diferentes partes do seu mecanismo e quais os papéis desempenhados por cada uma delas. Após finalizar a leitura deste capítulo, o leitor será capaz de criar seus próprios problemas de AR usando qualquer um dos dois conjuntos de ferramentas de criação disponibilizados pelo Barbell: o leitor de *scripts*, YAML que lê um arquivo com uma descrição textual da estrutura do problema de aprendizado/ambiente (incluindo a especificação da dinânamica do ambiente, estrutura física do agente, ações disponíveis ao agente, e função recompensa), ou as funções

disponibilizadas pela API do projeto Barbell, que, se chamadas diretamente, executam as mesmas funções. Todas as entidades envolvidas em um problema de AR (como o agente e as partes que o compõem, bem como o ambiente e seus objetos) podem ser criados através da chamada de comandos específicos fornecidos pela API ou através de um *script* escrito em YAML, que define a estrutura do projeto (sob o formato de uma estrutura que combina vetores associativos e listas de elementos, como explicado na seção 2.6 essa parte está confusa. O que tu quer dizer com projeto, exatamente? é a especificacao de um ambiente especifico? o conjunto de arquivos necessarios pra fazer isso? é o script yaml? nao definiu o que signfica projeto. A descricao de que é implementado com vetores associativos e listas tambem ta detalhada demais e fora de lugar, aqui nessa etapa do texto, aonde tu ainda ta recebem comecando a explicar o que é. O leitor nem sabe direito o que tu fez e tu ta explicando que internamente é um vetor associativo) e as informações a respeito de todas as partes que cumprem algum papel na simulação. Neste capítulo, portanto, será dada uma breve porque breve? esse é o capto principal explicação sobre a estruturação de um projeto projeto=ambiente? utilizando-se do *framework* Barbell, bem como quais comandos devem ser executados (ou quais linhas devem ser incluídas no *script*) para que sejam criados o agente, o ambiente e todas as demais partes do problema.

4.1 Características básicas do funcionamento sistema

4.1.1 Ciclo de aprendizado

O sistema qual sistema? o barbell implementa esse ciclo? ou ele gera ambientes com os quais um agente de AR pode interagir, ao ser utilizado (por outros softwares) numa tarefa de aprendizado que siga esse ciclo? segue o formato do ciclo de aprendizado via interação com o ambiente descrito na figura 2.1. No início de cada episódio, o agente e os demais elementos dispostos pelo ambiente são inicializados, em posições pré-estabelecidas ou de maneira aleatória, dentro de regiões demarcadas no espaço através do *script* que define o cenário. Em cada iteração, o agente deve executar uma ação, e todas as ações possíveis também devem ser definidas no documento. O resultado da leitura do documento que define a simulação é um objeto da classe `Barbell`. A este objeto, através de funções que recebem outras funções como argumento, podem ser fornecidas as funções responsáveis pela tomada de ação do agente, pelo cálculo da recompensa e por decidir se o episódio chegou ao final ou não. O fornecimento destas funções ao objeto resultante

da leitura do *script* é opcional, mas altamente recomendado, visto que, desta maneira, automatiza-se todas as verificações necessárias em uma iteração ou em um episódio de aprendizado por reforço especificar a funcao que escolhe acao, a funcao que determina a recompensa, etc, essas coisas sao opcionais? mas o codigo que o gym usa nao exige que tenha uma funcao de recompensa, que tenha algo equivalente ao takeAction, etc? Nessa parte aqui, precisa ter uma introducao melhor. Precisa comecar fazendo um link pra como o gym funciona (que deveria ter aparecido no capto anterior): quais metodos ele exige que tu implemente pra criar um agente e um ambiente que sejam compativeis com a API que ele especifica. Aí tu precisa dizer algo tipo, nós vamos propor uma linguagem de alto nível para descricao das caracteristicas de um ambiente e de sua dinamica, e tambem das partes fisicas que irao compor um agente que ira operar nesse ambiente. Nesse capto iremos descrever que tipos de caracteristicas de um ambiente e de um agente poderao ser descritas pelo framework sendo proposto, assim como discutir o metodo pelo qual implementamos o processo de traducao/compilacao da linguagem de alto-nivel proposta para codigo de mais baixo-nivel, responsavel pela implementacao da simulacao, utilizando um motor de fisica (qual motor de fisica? no capto anterior tambem nao discutiu isso; ver email onde eu tinha falado sobre a necessidade de discutir esse trade-off entre implementar tudo do zero com um motor de fisica, e ter flexibilidade infinita, vs usa um framework pra geracoa automatica de simulacoes, que da menos flexibilidade mas gera codigo automatico e compativel com gym) . Um exemplo pode ser visto na seção **??**, onde uma classe, responsável pelas observações do estado da simulação e pelas tomadas de ação do agente, tem métodos específicios para estas tarefas e todos eles são fornecidos à biblioteca antes de iniciar-se o experimento (ver o código em Python da subseção **??**).

4.1.2 Representação dos elementos envolvidos

Existem no Barbell, resumidamente, duas grandes figuras envolvidas em representar, de maneira complementar, todos os elementos de um cenário de AR: a biblioteca de física e a biblioteca gráfica. A biblioteca de física é responsável pela criação do espaço físico bidimensional e pela representação do agente e dos objetos do ambiente neste espaço, bem como pela aplicação de forças newtonianas em cada um dos corpos presentes no cenário. A simulação da interação entre os corpos, no que diz respeito às leis da física de Newton — referimos-nos aqui a colisões e atrito, por exemplo — também é um papel desempenhado pela biblioteca de física. Foi percebido, observando o projeto projeto? api?

Gym, que este, quando se trata de cenários onde robôs são representados em duas dimensões, faz uso da biblioteca PyBox2D, um *port* para a linguagem Python da biblioteca de física Box2D, que trata da representação de corpos rígidos em um espaço bidimensional. frase confusa. O Gym nao exige que pybox2d seja usado, ele é só uma api. tem que dizer que dentre os ambientes prontos que vem com o pacote, a maioria das implementacoes que sao compativeis com a api do gym usam uma biblioteca chamada pybox2d pra fazer a simulacao de fisica. embora outras sejam possiveis, desde que o codigo siga a interface, tu, nesse trabalho, escolheu criar o barbell de forma que ele gere codigo pybox2d automaticamente, a fim de fazer uso dessa biblioteca amplamente utilizada por outras pessoas da comunidade etc etc A biblioteca gráfica, por sua vez, é responsável por definir como se dará o processo pelo qual objetos, representados por pontos no espaço, serão desenhados na tela por meio de *pixels* ate agora nao tinha falado nada sobre o barbell tambem fazer algo em relacao a visualizacao. Era só sobre pegar a especificacao de alto-nivel de ambiente/agente e transformar pra codigo que gerasse codigo pra implementar a simulacao fisica dessas coisas. No capto anterior tambem nao falou nada sobre como o gym lida com visualizacao. Precisa dizer, na intro, entao, que o barbell gera nao só codigo pra implementar em linguagem de mais baixo nivel (o motor de fisica propriamente dito) o ambiente/agente, mas tambem cria codigo que implementa a visualizacao da simulacao, etc. Na hora de falar isso, tem que motivar essa necessidade, tambem, falando que, assim como a simulacao em si, que da trabalho escrever manualmente, a mesma coisa acontece pra escrever pra fazer visualizacao. Uma espécie de tradução é necessária, e fica a cargo de quem está desenvolvendo a API qual api? gym? a especificacao da linguagem de alto-nivel para descricao do ambiente? de fazer as duas bibliotecas trocarem informações entre si. No caso do Barbell, não foi diferente: notou-se uma diferença bastante expressiva na maneira que cada uma das duas bibliotecas representava os seus objetos e uma espécie de função de tradução teve de ser escrita para que a representação de um objeto no espaço físico fosse transformada em uma representação de um desenho formado por *pixels* na tela. O sistema de coordenadas da biblioteca de física segue a notação usada pelo sistema de coordenadas no plano Cartesiano: o eixo das abscissas cresce à medida que se avança para a direita, ao mesmo tempo em que as coordenadas crescem na medida em que se avança para cima. O sistema utilizado pela biblioteca gráfica para representar os objetos na tela, entretanto, é fundamentalmente diferente: o eixo das coordenadas cresce à medida em que se avança em direção à parte mais inferior da tela, de forma que os quatro pontos que representam as quatro extremidades de uma tela de 640 *pixels* de

largura por 480 de altura, a começar pelo canto superior esquerdo e se avançando no sentido horário, são: $(0, 0)$, $(640, 0)$, $(640, 480)$ e $(0, 480)$. A transformação empregada pelo Barbell para traduzir coordenadas cartesianas para o sistema empregado pela biblioteca gráfica acontece de tal forma que o ponto $(0, 0)$ da biblioteca de física é representado no canto inferior esquerdo da tela, com as coordenadas horizontais crescendo à direita e as verticais crescendo para cima.esse tipo de detalhe, de como as coordenadas sao mapeadas entre a simulacao da fisica e a visualizacao, ta fora de lugar aqui. tu nem explicou ainda o que é o barbell, exatamente, e como tu especifica propriedades do ambiente/agente/visualizacao, e ja ta falando sobre como internamente é feito o mapeamento de coordenadas dos objetos pra coordenadas na tela. ISso tem que ir numa secao mais pra frente falando especificacao de como o barbell gera codigo pra implementar essa comunicacao entre as duas bibliotecas de mais baixo nivel prais quais ele gera codigo: box2d e a especifica de visualizacao que tu escolheu

4.2 Estrutura compreendida pelo Barbell

Conforme explicado em capítulos anteriores e sumarizado na figura 2.1, um problema de AR possui entidades que cumprem papéis diferentes porém igualmente fundamentais para o decorrimento de uma tarefa de aprendizado. Cada uma dessas entidades é compreendida de maneira diferente pelo sistema descrito neste trabalho e, portanto, requer sua própria maneira de ser definida essa ultima frase ta vaga. que que isso quer dizer, exatamente?. Para facilitar a escrita do *script*, todos os comandos necesśarios para a modelagem dos elementos de um cenário de aprendizado por reforço foram reunidos sob três grandes grupos. Conforme o que foi desenvolvido na seção 2.6, podem ser definidos pares chave/valor em um arquivo YAML, onde um valor pode ser uma lista de valores ou ainda outro dicionário. Estes três grupos, apesar de serem representados apenas por chaves no arquivo, agrupam, abaixo de si na estrutura do *script*, configurações necessárias para o completo funcionamento da simulação. Os três grupos são:

- **DOMAIN**: neste grupo, são reunidas configurações a respeito da interface gráfica que desenha na tela o resultado da simulação. Detalhes técnicos como taxa de atualização da tela, formato da janela onde são desenhados os objetos e título da janela são definidos aqui. Todo valor configurável nesta seção do código possui seu valor padrão e, portanto, a configuração destes somente se dará em casos avançados,

onde quer-se que o resultado produzido pela interface gráfica seja diferente daquele que é produzido normalmente.

- **AGENT**: Aqui são feitas as definições do agente. As partes físicas que o compõem, por exemplo, são detalhadas nesta seção. Cada parte exige que informações específicas a seu respeito e, portanto, para cada uma das partes que constituem o agente, uma série de definições a respeito das suas propriedades físicas (formato, densidade, atrito em suas extremidades, *etc.*) devem estar presentes nesta parte do *script*. Aqui também são definidas as ações possíveis do agente e como as partes que o formam se conectam umas com as outras e com os objetos do ambiente.
- **ENVIRONMENT**: Aqui são feitas as definições a respeito de tudo que está inserido no cenário do problema de AR a ser simulado e e que não faz parte do agente. Definições globais como gravidade, por exemplo, deverão ser feitas nesta seção. Além do agente, estão inseridos no ambiente objetos com os quais ele pode interagir e é aqui, também, que estes devem ser definidos.

Para definir elementos pertencentes a um destes três grupos é necessário que o elemento seja declarado internamente, em termos de identação, ao marcador que indica o nome do grupo. Portanto, a estrutura básica de um *script* Barbell possui o seguinte formato:

```
1  DOMAIN:
2    #definições pertencentes ao grupo DOMAIN
3  AGENT:
4      #definições pertencentes ao grupo AGENT
5  ENVIRONMENT:
6      #definições pertencentes ao grupo ENVIRONMENT
```

Nas próximas seções, será descrito, com o auxílio de exemplos, como devem acontecer as definições dos elementos pertencentes a cada um destes grupos. Após o entendimento do restante deste capítulo, o leitor estará apto a desenvolver seus próprios cenários de aprendizado por reforço. o tcc nao é um manual de uso de um software, entao nao precisa enfatizar esse tipo de coisa. O objetivo desse capto é explicar o que tu fez/decidiu incluir na linguagem de especificacao, e porque; p.ex., da pra definir agentes com partes definidas por poligonos arbitrarios? caso nao, tu precisa discutir esse tipo de coisa, dizer que tu escolheu uma linguagem de especificacao com tais e tais caracteristicas e tal poder de expressao, porque tu fez uma analise dos problemas que vêm modelados por default no gym, e tu viu que uma linguagem com poder de expressao como tu definiu seria suficiente

pra expressar a dinamica de 90% daqueles ambientes, ou algo assim. Esse tipo de discussao precisa aparecer, pra explicar o que tu fez e porque isso é suficiente/uma vantagem sobre o que ja existe

4.3 *Domain*

Conforme dito anteriormente, esta é a parte do *script* onde se realiza a configuração dos valores usados pela parte gráfica da biblioteca qual biblioteca grafica? quais sao normalmente usadas por ambientes implementados no gym? quais metodos pra visualizacao o gym exige que tu implemente, caso siga a API deles? algum? nenhum? Isso precisa estar discutido. O teu framework vai gerar codigo pra alguma biblioteca grafica especifica? Qual? Porque tu escolheu ela?. Nesta seção, as configurações se dão através de pares chave/valor onde as chaves podem ser as seguintes:

- **width**: requer um valor que indica a largura em *pixels* da tela onde o programa será exibido (caso ele seja). É um valor inteiro positivo e o valor padrão é 640.
- **height**: chave que exige um valor que indica a altura em *pixels* da tela onde o programa será exibido (caso ele seja). Deve ser informado um valor inteiro positivo e o valor padrão é 480.
- **target_fps**: é a configuração que define a taxa de atualização da tela onde será mostrado o programa, indicando quantos quadros por segundo serão exibidos. Deve ser informado um valor inteiro positivo e o valor padrão é 60.
- **caption**: determina o texto que vai na barra superior da janela, que normalmente é usado para dizer qual o programa em execução, em sistemas operacionais como Windows e Linux. Qualquer texto é suportado, e o padrão é a frase "another Barbell project".
- **background_color**: define a cor do plano de fundo da simulação, com a qual será pintado todo *pixel* que não corresponder ao desenho de nenhum objeto. É uma lista de quatro números reais no intervalo $[0, 255]$ que representam uma cor no formato RGBA. O padrão é a tupla (0, 0, 0, 0).
- **draw_joints**: chave que requer um valor booleano que determina se as juntas entre as partes físicas do agente, ou entre estas e os objetos do ambiente, serão desenhadas. Algumas juntas, como uma junta de distância, por exemplo (a ser discutida na Seção XYZ), podem ser representadas por uma linha, para demarcar graficamente

que há uma junta ali. Em alguns casos, entretanto, pode não ser interessante que as juntas sejam desenhadas e, portanto, a possibilidade de representar as juntas graficamente através de linhas pode ser configurada. O valor padrão é *false*, ou seja, o programa não representa normalmente as juntas.

- **joint_color**: requer uma lista de quatro valores no intervalo $(0, 255)$ que define a cor (também em formato RGBA) que será usada para desenhar as linhas correspondentes às juntas. O valor fornecido aqui é ignorado caso as juntas não sejam desenhadas. valor padrão é a tripla (255, 63, 63, 0).
- **joint_width**: define o valor que define a espessura, em *pixels* das linhas que representam as juntas. É representado por um valor numérico real positivo, e o padrão é 1.
- **ppm**: todos os elementos de uma simulação de aprendizado por reforço, no Barbell, têm suas dimensões definidas em metros. Na transformação de um objeto com, por exemplo, dois metros de largura, em um desenho na tela com dimensões determinadas em *pixels*, algum critério deve ser usado. PPM (sigla para *pixels per meter*, ou *pixels* por metro), é o número que define a escala dos objetos no momento que estes devem são desenhados. É um valor numérico real, positivo, e cujo valor padrão é 50.

Abaixo, um exemplo de definição de um DOMAIN em um *script* YAML feito para o Barbell:

```
DOMAIN:
  width: 800
  height: 600
  caption: My Barbell project
  target_fps: 60  # valor padrão
  background_color: [255,255,255, 0]
  draw_joints: true
  joint_color: [150, 150, 150, 0]
  joint_width: 2
  ppm: 75
```

4.4 *Agent*

De estrutura um pouco mais complexa do que o DOMAIN, o grupo AGENT possui, sob a sua estrutura, outros três subgrupos, que também são chaves de pares chave/valor. Cada uma dessas chaves requer a declaração de uma lista de elementos, visto que as chaves representam grupos de objetos semelhantes. Estes grupos são:

- **PARTS**: nesta parte da estrutura, é declarada uma lista de partes físicas [sempre enfatiza que sao partes fisicas, porque um agente, em IA, nao é só a especificacao do corpo/capacidades dele, mas tambem do algoritmo de aprendizado que ele usa; mas tu nao usa o Barbell pra descrever ou implementar metodos de aprendizado, só pra fazer a criacao do codigo de simulacao dos efeitos de interacao do agente com seu ambiente] que constituem o agente, e as informações a respeito de cada uma delas. aqui precisa dizer que, mais adiante, na secao XYZ, tu vai especificar quais sao os tipos de partes físicas que podem compor um agente; que, resumidamente, elas correspondem a diferentes tipos de poligonos que se pode combinar atraves de juntas para especificar a forma e propriedades de diferentes partes do corpo do agente
- **JOINTS**: nesta seção, são definidas as junções entre partes diferentes do agente ou entre uma parte do agente e um objeto inserido no ambiente. Juntas são uma espécie de limitação física que, por exemplo, pode manter a distância de dois objetos fixada a um certo valor. Mais sobre juntas na seção 4.4.2.
- **ACTIONS**: nesta seção, definem-se as ações que podem ser tomadas pelo agente, sob a forma de forças físicas que atuam sobre partes dele, para dá-las movimento. daria pra criar uma acao do tipo 'dar tiro', que nao muda nada no corpo do agente, mas que aplica uma forca a um objeto (bala) do ambiente? caso sim, clarificar aqui, porque senao parece que o tipo de acao é só de um tipo—só acoes sao forcas sobre o proprio corpo do agente; caso nao, clarifica tambem, e explica porque tu decidiu simplificar dessa forma Toda decisão que o agente toma resulta na aplicação de uma força física em um ponto específico de uma determinada parte dele. As ações possíveis devem ser listadas nesta porção do *script*.

4.4.1 *Parts*

Nesta seção do *script*, interna ao grupo AGENT, deverá ser criada uma lista de todas as partes que constituem o agente, da mesma maneira que foi criada uma lista das habilidades possuídas pelos empregados no exemplo da seção 2.6. Cada item da lista é, em si, um par chave/valor, onde a chave é o nome da peça (um conjunto único de letras, números e o símbolo _) e o valor é um conjunto de pares que relaciona suas configurações e seus valores associados a cada uma delas. No exemplo abaixo, um trecho de um *script* mostra como deve ser a seção PARTS. Nela, há a palavra chave PARTS seguida de uma lista de duas peças chamadas PART_A e PART_B.

```
PARTS:
  - PART_A:
      # definições das características da parte PART_A
  - PART_B:
      # definições das características da parte PART_B
  # ...
```

Há três tipos de partes diferentes: *box*, *polygon* e *circle*. Todas elas possuem características configuráveis que são próprias ao seu tipo ou que são comuns a todos eles. Sua principal diferença é o formato do objeto que criação: partes do tipo *box* são retangulares, ao passo que partes do tipo *circle* são circulares e partes do tipo *polygon* podem assumir qualquer formato. Fica óbvio aqui que podem ser criadas partes retangulares do tipo *polygon* mediante o fornecimento de vértices que definam uma área retangular; foi tomada a decisão, entretanto, de fornecer esse atalho aos usuários do Barbell para que seja ainda mais fácil definir partes de formato simples. As chaves que podem ser contidas no dicionário que define uma parte são as seguintes:

- **type**: conforme dito anteriormente, esta configuração pode assumir três valores: "box", "circle"e "polygon". O valor "box"define que a parte será retangular, com tamanho definido na configuração "box_size". Caso o valor informado seja "circle", a parte será circular, e é necessário informar o seu raio através da palavra chave **radius**. Caso seja informado o valor "polygon", uma lista de vértices deverá ser informada na palavra-chave "vertices".
- **box_size**: usada quando a parte é do tipo "box". Deve ser informada, junto com essa palavra-chave, uma tupla (a, b) de dois números reais positivos, respectivos

à altura e à largura da peça, em metros, a contar do seu ponto central. A parte resultante será, então, um objeto de formato retangular e de altura $2a$ e largura $2b$.

- **radius**: palavra-chave usada quando a peça é do tipo "circle". Especifica o raio da peça, em metros.
- **vertices**: caso a peça seja do tipo "polygon", deverá ser informada, através desta palavra-chave, uma lista de pares que definem, em coordenadas locais, os vértices da parte. Vértices ligados por arestas devem estar adjacentes na lista informada ou serem o primeiro e último elementos.
- **initial_position**: nesta configuração, deve ser informada posição inicial da parte, em coordenadas globais qual o sistema de coordenadas do mundo? ele é centrado no 0,0? depois na hora de mapear isso pra visualizacao, o que acontece com o que cai fora da janela? ou ele sempre da um zoom out pra caber tudo?, através de uma tupla de dois números reais positivos. Também pode ser informada a palavra "random", para que, a cada episódio, a peça seja inicializada em uma posição diferente. Neste caso, devem ser informadas outras duas configurações: "x_range"e "y_range".
- **x_range**: no caso de uma peça que é iniciada em uma posição aleatória a cada novo episódio, deve ser informado aqui uma tupla de dois valores (x_1, x_2) com $x_1 \leq x_2$, que indica o intervalo no qual a posição x da peça irá variar.
- **y_range**: mesma coisa que a palavra-chave acima, porém para a coordenada y da peça.
- **angle**: aqui deve ser informado um valor real positivo que simboliza a rotação da peça, em graus, ou a palavra "random"para randomizar o ângulo no qual a peça é rotacionada no instante que é criada no começo de um episódio. O valor padrão é zero.
- **angle_range**: intervalo no qual o ângulo da peça vai variar, no instante que é criada, caso ele tenha sido definido como aleatório. É um par (α_1, α_2), com $0 \leq \alpha_1 \leq \alpha_2 \leq 360$.
- **static**: palavra-chave que define um valor booleano, que diz se um corpo é estático (não sofre nenhum tipo de força, permanecendo parado) ou dinâmico. É padrão um corpo ser dinâmico para partes de um agente, e estático para objetos do cenário.
- **density**: nesta configuração, deve ser informado um valor real positivo que simboliza a densidade do corpo, em gramas por centímetro cúbico. Valor padrão é 0,1.

- **friction**: aqui, deve ser informado um valor real positivo que simboliza o coeficiente de atrito da peça. O valor padrão é 0,25, que é próximo do coeficiente de atrito de um pneu no asfalto.
- **color**: aqui, deve ser declarada uma lista de quatro valores reais no intervalo $[0, 255]$, que definem a cor (em formato RGBA) com a qual será desenhada a peça. O padrão são os valores $(155, 155, 155, 0)$.

4.4.2 *Joints*

Nesta seção, ainda dentro do grupo de configurações referentes ao agente, deverá ser criada uma lista que definirá todas as juntas do sistema. Uma junta é uma espécie de conexão mecânica de algum tipo, que conecta uma parte do agente ou com outra parte dele ou com um objeto disposto no ambiente, o que é de extrema importância na hora de definir movimentos complexos que envolvem mais de uma parte por vez (braços mecânicos, por exemplo, como o da figura 2.2). no *script* YAML que será fornecido ao Barbell, as juntas podem ser de três tipos diferentes:

- **distance**: *distance joint*, ou simplesmente "junta de distância", é o tipo que mantém dois corpos (chamados de *anchors*, ou "âncoras") a uma distância fixa, de tal maneira que, se uma força for aplicada em um deles e isso resultar em um deslocamento, o outro corpo será deslocado junto com ele. Uma junta de distância precisa de dois pontos de referência, um em cada corpo. Com a junta definida, a distância se manterá fixa entra a âncora de um corpo e a âncora do seu par.
- **revolute**: semelhante à uma junta de distância uma *revolute joint*, ou "junta de revolução", une as âncoras de dois objetos em um único ponto, fazendo com que um deles (ou os dois) possam girar em torno desse ponto. É importante adicionar que, uma vez que dois objetos estejam unidos por uma junta de revolução, eles passam a não colidirem mais um com o outro.
- **prismatic**: uma *prismatic joint*, ou "junta prismática", permite que seja traçado um eixo, paralelo a um ponto de um corpo e mantendo distância fixa a ele, sob o qual o outro corpo desliza. É útil em modelagens onde se há um movimento de vai e vem, como é o caso em elevadores, por exemplo.

O método para definir juntas é praticamente o mesmo usado para definir as partes do agente; juntas, todavia, não possuem nome. Sua estrutura, portanto, tem um nível a

menos de profundidade do que a estrutura das partes do agente. As palavras-chave usadas no dicionário que define uma junta são as seguintes:

- **type**: *string* que define o tipo da junta. Pode ser "distance", "revolute"ou "prismatic".
- **anchor_a**: palavra-chave que define a âncora A da junta, ou seja, a parte que ela irá conectar.
- **anchor_b**: palavra-chave que define a âncora B.
- **anchor_a_offset**: juntas de revolução, quando criadas, levam em consideração o ponto de coordenadas globais que corresponde ao ponto $(0, 0)$ das âncoras no momento de sua criação. Para prender uma junta em qualquer ponto que não o ponto central dos corpos, é necessário informar um deslocamento que será aplicado em relação ao centro do objeto. Aqui, é informado o deslocamento do ponto da âncora A.
- **anchor_b_offset**: palavra-chave usada para informar o deslocamento do ponto onde a junta irá atuar na âncora B em relação ao seu centro.
- **axis**: nas juntas prismáticas, é criado um eixo em relação à âncora A no qual a âncora B irá deslocar-se. Utilizando-se da palavra chave "axis", deverá ser definido um vetor (preferencialmente unitário) que definirá a direção do eixo afixado à âncora A.
- **enable_motor**: há a opção, no caso de juntas prismáticas, de ligar-se uma espécie de motor, que aplica uma força à ancora B, que desloca-se sobre ele. Através de um valor booleano (cujo valor padrão é *false*), mantém-se este motor ligado ou desligado.
- **motor_speed**: no caso de juntas prismáticas, esta é a aceleração aplicada à âncora B quando a configuração "enable_motor"está ligada.
- **max_motor_force**: aceleração máxima que a âncora B pode receber, no caso de uma junta prismática.
- **enable_limit**: configuração que recebe um valor booleano (seu valor padrão é falso) que define se os limites do eixo de deslocamento da âncora B estarão ativos. Se sim, fica estabelecido um limite ao qual a âncora B pode se deslocar pelo eixo, em ambos os sentidos.
- **upper_translation**: limite, em metros, que a âncora B de uma junta prismática pode se deslocar no sentido a favor ao do vetor que definiu o eixo.

- **lower_translation**: limite, em metros, que a âncora B de uma junta prismática pode se deslocar no sentido contrário ao do vetor que definiu o eixo.

Fica bastante evidente, ao observar a lista acima, que o grau de complexidade de juntas prismáticas é muito maior do que o de juntas de distância ou de revolução. Um exemplo que engloba praticamente todas as configurações possíveis de uma junta prismática é dado na seção **??**.tem varios labels desses que nao estao definidos com o comando label logo depois da declaracao da section ou subsection correspondente, e ai o comando ref nao funciona. tem que arrumar

4.4.3 *Actions*

ACTIONS é a seção do *script* YAML onde são definidas as ações que poderão exercidas sobre as diferentes partes do agente precisa discutir aqui que as tuas acoes, entao, sao necessariamente acoes fisicas sobre o proprio agente. Nao pode ter uma acao do tipo 'comer', que diminui em um o contador de comida que o agente tem, ou algo assim? i.e. uma acao cujo efeito a pessoa informa atraves de preenchimento de codigo de alguma funcao stub que tu cria, e que afeta alguma coisa interna ao agente (tipo um contador), mas nao afeta nenhuma parte fisica necessariamente? Caso nao de pra fazer isso diretamente atraves de especificacao no yaml, discutir aqui ou depois que é uma limitacao da linguagem, e falar como a pessoa poderia pegar o codigo gerado pelo barbell e estender manualmente pra especificar esse outro tipo de acao. É importante ressaltar que os componentes de um agente e os objetos dispostos no cenário são iguais em sua composição — ambos são corpos físicos com as mesmas propriedades e características — mas é somente sobre os componentes do agente que podem ser aplicadas forças. Cada decisão que o agente toma é relativa a uma força que é aplicada sobre ele: diferentes formas de mover-se requerem forças de intensidade variável e aplicadas em lugares diferentes.

Forças são representadas por vetores bidimensionais, que dizem qual a direção e sentido que a força será aplicada. Há três tipos de força no Barbell: "local", "global"e "torque". Forças locais e globais dizem respeito a qual sistema de coordenadas é levado em consideração na hora de aplicar a força. Forças locais utilizam as coordenadas locais de um objeto como orientação; uma força de vetor $(2, 0)$ aplicada em uma caixa, por exemplo, impulsionaria uma caixa para cima. Se a caixa fosse objeto de outra força, que a fizesse girar $180°$, e a força local fosse aplicada nela de novo, ela já não iria ser im-

pulsionada para cima, mas em direção ao chão, pois, apesar de ela estar em uma posição completamente diferente, suas coordenadas locais continuam as mesmas. Forças globais, entretanto, apontam sempre pra mesma direção independentemente da orientação do objeto: uma força global que aponta para a esquerda irá sempre apontar para a esquerda. Já forças do tipo torque, por sua vez, são aplicadas em objetos quando se quer que os mesmos rotacionem em volta do seu centro.

Para definir as forças que serão usadas na simulação, basta que seja definido no arquivo YAML uma lista interna à palavra-chave FORCES, com cada elemento da lista sendo um dicionário com as seguintes palavras chaves:

- **type**: define tipo de força, conforme especificado acima.
- **target**: especifica o nome da parte do agente que sofrerá a ação desta força, quando aplicada. A parte deve existir.
- **anchor**: define o ponto, em coordenadas locais, da parte do agente onde a força será aplicada.

IMPORTANTE: falta nesse capto descrever COMO tu fez o framework. Por enquanto tu só descreveu O QUE DÁ PRA FAZER com ele. Fala, p.ex., de como tu faz pra transformar o codigo no script yaml pra codigo pro box2d. Tem um mapeamente 1-pra-1 facil entre as secoes do script e metodos do box2d? tu implementou manualmente algum tipo de parser e compilador pra gerar codigo pro box2d e pro sistema de visualizacao? e o codigo que o processamento do teu script yaml corresponde a codigo implementando quais metodos do gym? tem algum metodo da API do gym que o teu sistema gera codigo, e que depois pode ser estendido manualmente pelo usuario, caso queira dar mais flexibilidade?

IMPORTANTE: A gente tinha discutido aquelas coisas tipo, como se define no barbell o que vai compor o estado do agente? e a funcao de recompensa? essas sao coisas que o gym exige que sejam implementadas/especificadas, mas nao estao discutidas

4.5 *Environment*

Duas coisas são feitas nesta seção do *script*: são definidos detalhes do ambiente — como a força da gravidade — e são definidos os objetos pertencentes ao ambiente. Aqui, as seguintes palavras-chave são usadas para realizar configurações:

- **gravity**: palavra chave utilizada para definir o vetor de força da gravidade, que será

aplicado a todos os objetos dinâmicos a cada passo da simulação.

4.5.1 *Objects*

Além da gravidade, são definidos no grupo ENVIRONMENT todos os objetos pertencentes a ele. A definição dos objetos é simples e é praticamente igual à definição de partes do agente, conforme documentado na seção 4.4.1, com a diferença de que objetos do ambiente não podem ser alvos de ações e a palavra-chave na qual eles serão declarados é OBJECTS, ao invés de PARTS.

5 EXPERIMENTOS

Foram realizados experimentos a fim de mostrar situações onde normalmente se faria uso de um simulador de física para, de maneira totalmente, modelar o agente e o ambiente onde ele atuaria. Os problemas abaixo descritos foram escolhidos usando-se como critério a proximidade de exemplos vistos na documentação de outros simuladores quais?. Assim, facilita-se a comparação entre o uso através do sistema documentado neste trabalho e outros *frameworks* já existentes quais? mujoco e mais algum? caso mais algum, teria que descrever tambem esse outro framework no capto anterior.

5.1 Cartpole

O cartpole talvez seja o exemplo mais usado para fins didáticos dentro da área de aprendizado por reforço. Nesse problema, existe um carrinho (também chamado de *cart*) que tenta, através de movimentos laterais, equilibrar uma vareta (chamada de *pole*), presa ao *cart* de alguma forma. Aqui, o ambiente é praticamente inexistente: a única coisa que pode existir além da dupla *cart-pole* é um trilho pelo qual o *cart* se desloca. A leitura do estado do sistema, portanto, irá retornar apenas informações referentes ao próprio agente: posição e velocidade atual do *cart*, tal como o ângulo atual do *pole* e a velocidade com que o ângulo está atualmente sofrendo alterações.

A recompensa é dada pelo tempo que o *cart* consegue manter a vareta em equilíbrio antes que esta caia ou que um limte de tempo seja atingido. A recompensa, então é igual ao número de iterações durante as quais o *pole* permaneceu em equilíbrio.

5.1.1 *Cartpole* no OpenAI cuidar pra chamar o Gym de maneira consistente; as vezes aparece openai gym, as vezes openai, as vezes só gym

O problema do *cartpole* no ambiente OpenAI é bastante simples, porém igualmente limitado. De acordo com a descrição disponível na página da OpenAI, apenas duas ações estão disponíveis: aplicar uma força de +1 ou -1 no *cart*, baseado em leituras que retornam o estado do agente. Os episódios iniciam com o *pole* na posição vertical e acabam quando o seu ângulo é maior do que 15 graus em relação à sua posição inicial.

As leituras do agente retornam um vetor quadridimensional com as seguintes in-

formações tem que ser um pouco mais preciso na descricao desse problema. Em quase todos artigos que usam o cartpole, eles falam tipo, o estado s do agente é um vetor com 4 features, posicao, velocidade, etc; a funcao de transicao é tal que a mudanca na posicao x, $\dot{x}$, é dada por tal formula; a funcao de recompensa R num estado s é $R(s)$=tal coisa, onde tal coisa mede o angulo do pole. Coisas desse tipo. Aí diz que implementar o cartpole exige que ou tu implemente essas equacoes que descrevem a dinamica do pole de forma manual, ou entao que tu modele as propriedades fisicas do pole representado naquelas equacoes diretamente num motor de fisica, e deixar ele simular a dinamica das euqacoes. De qualquer forma, a descricao abaixo do que tu chama de 'informacao' precisa ser mais formal, dizendo que essas informacoes sao as features que compoem o estado s do agente, e que o motor de fisica implementa a funcao de transicao que diz como o estado do cart muda dependendo da acao, etc:

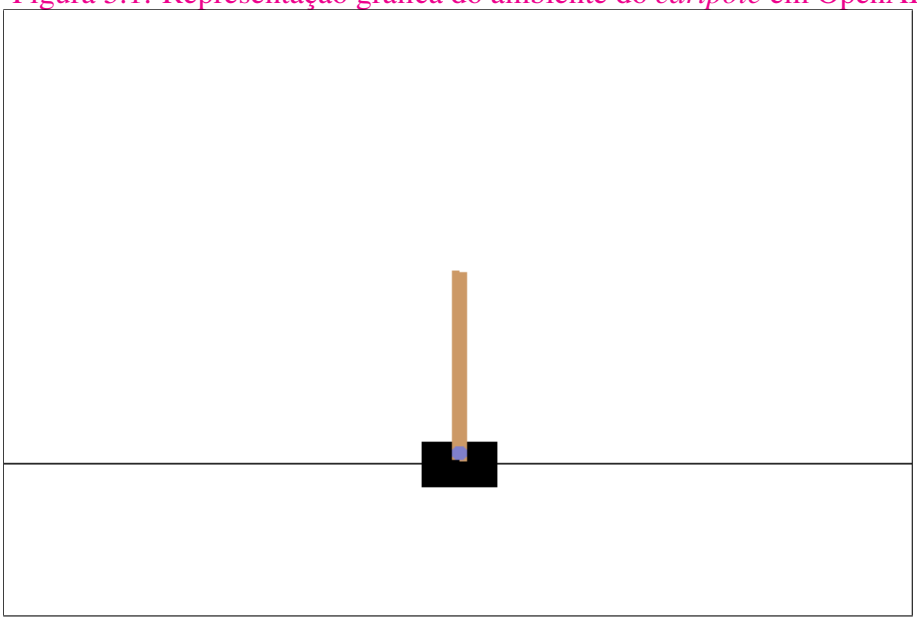
posição	**informação**	**mínimo**	**máximo**
0	posição do *cart*	-2.4	2.4
1	velocidade do *cart*	$-\infty$	$+\infty$
2	ângulo do *pole*	~$-41.8°$	~$41.8°$
3	velocidade da ponta do *pole*	$-\infty$	$+\infty$

O programa, portanto, está restrito ao ambiente fornecido pela OpenAI e deve operar de acordo com as regras estabelecidas por ele, sem que haja a possibilidade de alterá-las minimamente porque nao? como o cartpole ta implementado no gym? usando algum motor de fisica, ou usando essas equacoes classicas que eu mencionei no comentario anterior? caso usando um motor de fisica, qual? Descrever. E independente de qual seja a maneira, porque nao da pra modificar? Se for com motor de fisica, nao da pra facilmente modificar o peso/comprimento/etc do pole? massa do cart? forca que se pode aplicar em cada direcao?. O ambiente de desenho do simulador também já é fornecido pré-configurado ao programador e não fornece as ferramentas necessárias para modificar suas características (cores, tamanho da tela, etc).

5.1.2 *Cartpole* no *Barbell*

pro leitor conseguir saber se é mais facil no barbell, teria que ter visto como foi feito manualmente no ambiente que tem no Gym. Ver comentario acima, sobre discutir melhor exatamente como ele foi implementado (possivelmente mostrando o codigo,

Figura 5.1: Representação gráfica do ambiente do *cartpole* em OpenAI



mesmo, que nem tu fez com o barbell, abaixo, pra pessoa poder comparar)

No ambiente do *Barbell*, por este apresentar maior grau de liberdade ao programador, há a necessidade de se definir todas as partes do agente, todas as partes do ambiente e como elas se relacionam, antes de se dar início ao treinamento. Através de um *script* no formato YAML que é fornecido à ferramenta, o *framework* cria as partes do agente, inicializa o ambiente (também com suas partes específicas) e as relações entre as partes do agente e dos objetos do ambiente (através de juntas) a explicacao de como o framework funciona, de forma geral, tem que ir no capto anterior, aonde tu descreve o framework propriamente dito. O capto atual é sobre demonstracoes de como aplicar o framework pra recriar alguns ambientes de exemplo, e comparar o quao facil/dificil/flexivel é a implementacao via teu framework, vs frameworks alternativos que tu descreveu no capto de estado da arte. É tu tentando vender o peixe de que o teu framework é mais facil/tem vantagens, em relacao ao que ja existe. Todas as definições necessárias para o problema estão presentes no código a seguir:

```
1 \bruno{no capto onde tu vai descrever o teu framework, descreve todas e
2
3   AGENT:
4     draw_joints: false
5     joint_color: [255,0,0,0]
6     parts_color: [175,175,175,0]
7
```

```
  PARTS:
    - cart:
        type: box
        initial_position: [12, 9]
        box_size: [2, 1]
        color: [255,63,63,0]
    - pole:
        type: box
        initial_position: [12, 16]
        box_size: [0.1, 2.5]

  JOINTS:
    - connects: [pole, cart]
      type: revolute
      anchor_a: [0, -2.2]
    - connects: [floor, cart]
      type: prismatic
      anchor: [12, 5]
      axis: [1, 0]
      lower_translation: -3
      upper_translation: 3

  ACTIONS:
    - push_cart:
        type: local
        target: cart
        anchor: [0,0]
    - start_pole:
        type: local
        target: pole
        anchor: [0, 2.5]

ENVIRONMENT:
  gravity: [0, -10]
```

```
42      floor: none
43      OBJECTS:
44        - floor:
45            type: box
46            initial_position: [12, 1]
47            box_size: [12, 0.5]
```

No código, há a definição das partes do agente (linhas 6-15), dos objetos presentes no cenário (linhas 41-45) e das juntas que as conectam. Uma junta é necessária para conectar o *cart* ao *pole* (linhas 18-20) e outra junta, do tipo prismática, cria um eixo paralelo ao chão que torna possível que o *cart* deslize sobre ele (linhas 21-26). O códido do programa em *python* que se responsabiliza de rodar os testes tem a responsabilidade, portanto, de fornecer a descrição YAML acima e iniciar cada um dos episódios, fornecendo funções que calculem a recompensa ao final de cada episódio, que determinem se um episódio chegou ao fim e que escolham qual ação o agente irá tomar, adicionando, ao início de cada episódio, uma força aleatória ao *pole* que determinará o comportamento do agente naquele episódio. O código que executa essas ações está explicitado abaixo.

```
1  ai = CartpoleIntelligence()
2  cartpole = barbell.from_file(os.path.join(
3                    os.path.dirname(__file__),
4                    'cartpole.yaml')
5                    )
6  cartpole.set_action_function(ai.action_function)
7  cartpole.set_reset_function(ai.reset_function)
8  cartpole.set_reward_function(ai.reward_function)
9  while cartpole.running:
10     current_state = cartpole.step()
11     if current_state["current_iteration"] == 0:
12         cartpole.agent.start_pole((random.randint(-10, 10), 0))
```

5.2 Lunar Landing

Exemplo do lunar landing, também recheado de figuras pra encher bastante linguiça.

5.3 Robô que atira bolinha (arrumar nome melhor)

Aqui vai um outro exemplo que eu queria desenvolver que é parecido com aquele vídeo do robô que tu me mostrou uma vez. O robô pegava uma bola e tinha que acertar uma garrafa de plástico posicionada aleatoriamente na frente dele. Também recheado de figuras que é pra encher bastante linguiça.

6 CONCLUSÃO

"O meu é melhor do que qualquer outro"em linguagem acadêmica.

REFERÊNCIAS