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
2019

“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 que compreendem uma subárea da inteligência artificial onde um agente, inserido dentro de um ambiente, tenta solucionar um determinado problema através de uma sequência de ações. Cada ação resulta em uma recompensa, e é com base apenas no acúmulo destas recompensas que o agente deve se guiar em busca da melhor solução para o problema. Problemas de aprendizado por reforço exigem, portanto, que o agente desenvolva um comportamento capaz de encontrar a melhor ação a ser tomada em um dado momento, a fim de maximizar o valor total das recompensas recebidas.

Normalmente, o processo de busca por uma solução aceitável é bastante custoso, pois é exigido do agente que este avalie diversas sequências possíveis de ações, refinando sequências encontradas anteriormente e buscando outras sequências completamente novas. Para acelerar a avaliação de soluções encontradas e, portanto, o treinamento do agente, é comum o emprego de simuladores, que constroem virtualmente o ambiente e o agente nele inserido.

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.

Este trabalho descreve o processo de criação de um *framework* de uso simples e que produz cenários padronizados, compatíveis com a API do *Gym*, *software* que vem sendo adotado como padrão no que diz respeito a ferramentas de *benchmark* de algoritmos de aprendizado por reforço (AR). Na ferramenta proposta por este trabalho, cenários são descritos através de uma linguagem de especificação de alto nível, permitindo que simulações de problemas de AR sejam modelados de maneira eficiente e que o resultado produzido esteja de acordo com ferramentas amplamente utilizadas na área.

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

ABSTRACT

Reinforcement learning methods deal with problems that comprise a subarea of artificial intelligence where an agent, inside an environment, tries to solve a problem through a sequence of actions. Every action results in a reward, and it is based only in the accumulated sum of these rewards that the agent must guide itself in search of the best possible solution for the problem. Reinforcement learning problems require, therefore, that the agent develop a behavior able to find the best possible action to be taken at a given moment, in order to maximize the total value of the rewards.

Usually, the process of search for an acceptable solution is costful, because the agent is required to evaluate several possible sequences of actions, refining sequences previously found and searching for other entirely new sequences. To speed up the evaluation of the found solutions, and, therefore, the training of the agent, it is common the use of simulators, that build virtually the environment and the agent in it.

There is already several frameworks that allow the building of simulators with certain degree of fidelity and that do not have a steep learning curve. There is, however, a cost associated to the adoption of such frameworks: this cost is related to the time needed to understand the tools provided by the framework and to reproduce the problem's environment using them.

This work describes a framework of simple use and that produces standardized scenarios, compatible with the interface of *Gym*, a software that has been adopted as a standard on which concerns benchmark tools for reinforcement learning algorithms. By using the tool proposed by this work, one can describe scenarios through a specification language, allowing reinforcement learning simulations to be modeled efficiently and also guaranteeing that the produced results are compatible with tools that are broadly used in the field.

Keywords: Electronic document. LaTeX. ABNT. UFRGS.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

IA	Inteligência Artificial
AR	Aprendizado por Reforço
MDP	Processo de Decisão de Markov
POMDP	Processo de Decisão de Markov Parcialmente Observável

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, modificando-o através delas. Não há, entretanto, nenhum mecanismo que permita ao agente compreender se uma determinada ação o deixa mais próximo de resolver o problema ou se ela nada influencia neste objetivo. A única maneira que o a gente tem de avaliar uma ação tomada em um determinado momento se dá através da noção de *recompensa*, que nada mais é do que um número escalar que é informado depois de cada ação tomada por ele 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, com base na situação do ambiente em um determinado momento, o leve a tomar uma série de decisões que resulte em uma soma máxima de recompensas. Esta busca pelo melhor comportamento possível normalmente é feita através de uma combinação entre ajustes na melhor política encontrada até um determinado momento (*exploitation*) e a busca de comportamentos completamente novos (*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 (THORNDIKE, 1898). 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 animais eram trancados nas gaiolas, eles deveriam explorá-las e, principalmente, interagir de forma autônoma com o interior da gaiola até que, por si mesmos, encontras-

sem 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 qual era 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 (MINSKY, 1954). 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 no qual um agente, ao interagir com o ambiente no qual ele encontra-se inserido, buscando resolver um problema, executa uma ação e recebe uma recompensa sobre a ação tomada, corrigindo seu comportamento de acordo com a recompensa recebida e com as modificações impostas ao ambiente pela ação, sempre de forma a maximizar o valor relativo à soma de recompensas recebidas pela sua sequência de decisões. Soluções que envolvem aprendizado por reforço são empregadas, consequentemente, em situações onde problemas devem ser resolvidos através de uma série de ações, como quando deseja-se, por exemplo, ensinar um agente a disputar partidas de jogos de tabuleiro (como no xadrez, onde uma sucessão de jogadas pode levar à vitória) ou ensinar um robô a executar tarefas que exijam uma série de movimentos (como fazer com que um robô quadrúpede se

locomova em trote da maneira mais rápida possível e mantendo o equilíbrio).

Para facilitar o treinamento de um agente em um problema de aprendizado por reforço, 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 (NG, 2003), 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 intempérie; 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; cada um deles, entretanto, possui as suas próprias limitações: normalmente, o que se observa é uma espécie de compensação entre simplicidade e fidelidade, 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 (Brockman et al., 2016), 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

No capítulo a seguir, será apresentada a formulação matemática do problema de aprendizado por reforço e alguns dos seus principais métodos de solução serão expostos. Além disso, é definido o que é um simulador de aprendizado por reforço. No capítulo 3, são apresentadas as diferentes ferramentas que podem auxiliar na construção de um simulador de aprendizado por reforço. No capítulo 4, o Barbell, sistema introduzido por este trabalho, é apresentado. No capítulo 5, experimentos são conduzidos. No capítulo 6, conclusões e propostas para trabalhos futuros são apresentados.

2 CONCEITOS BÁSICOS

Neste capítulo serão apresentados os conceitos fundamentais para a compreensão do trabalho desenvolvido, iniciando-se pela formulação matemática do um problema de aprendizado por reforço através de Processos de Decisão de Markov, seguido pela visão geral do fluxo de um algoritmo de aprendizado por reforço, com o uso de exemplos quando conveniente. O capítulo também trata do uso de simuladores para a modelagem de cenários de aprendizado por reforço.

2.1 Definição formal do problema de aprendizado por reforço

Sendo o problema de aprendizado por reforço um exemplo de situação onde decisões devem ser tomadas em sequência, este pode ser formulado matematicamente como um Processo de Decisão de Markov (também referido pela sigla MDP, do inglês *Markov Decision Process*). Um MDP é uma ferramenta 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 — ou seja, as propriedades — deste ambiente. Os processos modelados são chamados "de Markov" (ou *Markovianos*) porque obedecem à *Propriedade de Markov*: o efeito de uma ação em um estado depende apenas da ação tomada e do estado atual; e são chamados de processos "de decisão" porque modelam a possibilidade de um agente de interferir no sistema através da execução de ações (BELLMAN, 1957).

Usando o conceito de *recompensa*, que é um valor escalar que representa a qualidade de uma ação tomada quando o processo se encontra em um determinado estado, buscar uma solução para um Processo de Markov significa encontrar uma *política* — que nada mais é do que um conjunto de regras de decisão que diz ao agente qual ação deve ser tomada de acordo com o estado do ambiente — que não só leve a um estado terminal do ambiente, mas que também siga algum critério de otimalidade (que maximize o valor total das recompensas recebidas, por exemplo).

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 chamado de modelo de transição, 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 $s \in S$, denotada $R(a, s)$.

2.2 Horizonte

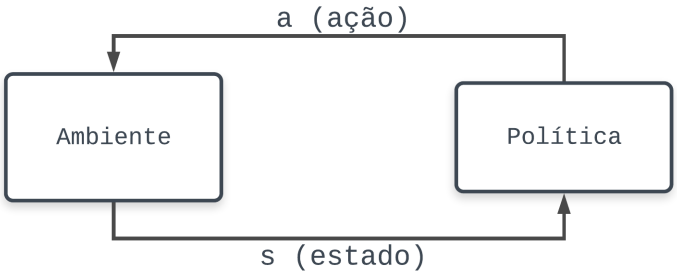
Em um MDP, o agente toma, a cada instante de tempo, uma decisão que resulta em uma transição entre estados e no recebimento de uma recompensa, com o objetivo de maximizar a soma das recompensas recebidas por uma série de ações tomadas. Sabendo que uma série de ações resulta em uma série de transições entre estados e no recebimento de uma série de recompensas, qual a influência que recompensas futuras devem ter, então, na tomada de decisão do agente, se comparadas com as recompensas que ele receberá imediatamente? Para isto existe, em processos de Markov, a noção de *horizonte*, um valor $T \in [1, \infty)$, que diz até que ponto no futuro, em uma série de ações a ser tomada pelo agente, as recompensas a serem recebidas exercerão influência em suas escolhas: para horizontes de tamanho 1, por exemplo, apenas a recompensa referente à sua próxima ação importa para o agente, independentemente de esta ação levá-lo a estados que no futuro resultarão em recompensas mais baixas — resultando em um agente que opera sob uma estratégia gulosa. Para horizontes de tamanho infinito, por outro lado, o agente tentará encontrar o valor máximo para a soma de todas as suas ações tomadas, ou seja, todas as recompensas que serão recebidas no futuro influenciam na decisão tomada pelo agente no presente.

2.3 Política de um MDP

Em um MDP, um tomador de decisões realiza uma leitura de um vetor s, que representa o estado atual do ambiente e executa uma ação a de acordo com uma política π, que é, basicamente, um conjunto de regras de decisão que diz qual ação deve ser tomada a cada estado. Uma forma simples de definir um conjunto de regras de decisão é através de um mapeamento direto de todos os estados possíveis para suas respectivas ações (na forma de uma função $\pi : S \mapsto A$). Uma política, portanto, é um conjunto que

reúne todas as regras de decisão de um MDP. A ação tomada pode trazer mudanças ao estado atual do ambiente, então uma nova leitura deste é realizada e uma ação é tomada, e assim sucessivamente, até que o problema seja resolvido — ou seja, até que um estado terminal $s_t \in S$ seja atingido. A figura 2.1 mostra o funcionamento do fluxo de tomadas de decisão em um sistema modelado por um MDP.

Figura 2.1: Funcionamento do processo de tomadas de decisão em um MDP.



O objetivo da resolução de um MDP é encontrar uma política $\pi(s) : s \rightarrow a$ que solucione o problema, ou seja, que diga ao tomador de decisões quais ações devem ser tomadas em cada estado $s \in S$ de forma que o problema seja resolvido. Dentre todas as políticas que resolvem um determinado problema, é considerada uma política *ótima* (denotada π^*) aquela que segue algum critério de otimalidade. Normalmente, o critério adotado é o valor esperado da soma das recompensas retornadas pela série de ações que o agente toma orientado por aquela política, ou seja, para uma política π^* este valor é máximo. Sua definição pode ser dada por

$$\pi^* = argmax_\pi \mathbb{E}[\sum_t^T R(s)|\pi] \tag{2.1}$$

que é a *recompensa esperada total*, ou por

$$\pi^* = argmax_\pi \mathbb{E}[\sum_t^T \gamma^t R(s)|\pi] \tag{2.2}$$

que é a *recompensa esperada descontada*, onde T é o horizonte (conforme descrito na seção 2.2) e $R(s)$ é a recompensa pela ação recomendada ao agente pela política π quando este encontra-se no estado s. Há ainda, na fórmula 2.2, um coeficiente $\gamma \in [0, 1]$ chamado de *fator de desconto* que serve tanto para garantir a convergência do valor da recompensa total esperada (em caso de horizonte infinito, com $T = \infty$), como para regular o impacto que recompensas mais imediatas exercem na política em relação a recompensas

que serão recebidas mais futuramente. O fator de desconto γ faz parte da *função valor* de uma política, que dá o valor esperado da recompensa para esta política e que é definida pela fórmula a seguir:

$$V^{\pi}(s) = R(s, \pi(s)) + \gamma \sum_{s' \in S} T(s, \pi(s), s') V^{\pi}(s') \tag{2.3}$$

A fórmula definida em 2.3 é usada na função que define o valor de uma ação a em um estado s considerando a recompensa imediata de a e considerando também que as ações subsequentes seguem a política π. A fórmula é denotada por Q e é definida por:

$$Q^{\pi}(s, a) = R(s, a) + \gamma \sum_{s' \in S} T(s'|s, a) V^{\pi}(s') \tag{2.4}$$

Mesmo definindo matematicamente o problema de aprendizado por reforço em através de um MDP e definindo os conceitos de função valor e política ótima, resta saber como buscar uma política que além de resolver o problema, seja uma política ótima. Há soluções exatas e aproximadas para o problema, e algumas delas serão discutidas a seguir.

2.4 Métodos de resolução de um MDP

Resolver um problema definido por um Processo de Decisão de Markov significa encontrar uma política π que, partindo de um estado inicial s, faça o agente eventualmente atingir um estado terminal s'. Além disso, normalmente não se quer apenas encontrar uma política π que resolva o problema, mas uma política ótima π^* que, além de oferecer ao agente um comportamento que resulte na resolução do problema, obedeça a algum critério de otimalidade, normalmente relacionado ao valor total da soma das recompensas recebidas.

Há diversos métodos que encontram tais políticas. Os mais simples, entretanto, envolvem soluções que utilizam programação dinâmica para encontrar a política de maior valor através da varredura de todos os estados do conjunto S e de todas as transições da função T, como é o caso no método de *iteração de valor* (SONDIK, 1971). Neste caso, o algoritmo é de *model-based* (baseado em modelos), uma vez que o a modelagem do problema (ou seja, os próprios componentes da tupla que define o MDP) é utilizada para encontrar a política que o soluciona.

Há casos, entretanto, que as transições de T e o conjunto S não estão disponíveis,

como é o caso de Processos de Decisão de Markov Parcialmente Observáveis (POMDP), utilizados para a modelagem de uma série de problemas onde o estado s é oculto ao agente (CASSANDRA, 1998). Quando o aprendizado se dá exclusivamente a partir da observação das recompensas recebidas a partir das ações tomadas em cada estado, se diz que o método é livre de modelos (*model-free*, em inglês), uma vez que a busca por um comportamento ótimo se dá mais explicitamente por métodos que envolvem tentativa e erro. Dos métodos livres de modelo, o mais conhecido se chama *Q-Learning*, descrito a seguir.

2.4.1 Q-Learning

Há situações onde é praticamente impossível (ou ao menos excessivamente custoso) buscar uma política ótima para um MDP através de um algoritmo baseado em modelos, em parte porque o espaço de ações e estados é muito grande, contribuindo para o custo computacional da busca por uma solução, e em parte porque algoritmos do tipo assumem que o agente possui conhecimento sobre o domínio no qual está atuando, ou seja, é presumido que o agente sabe de antemão as transições entre estados.

Para resolver este problema, há um algoritmo alternativo chamado Q-Learning (WATTKINS, 1989) que, ao invés de buscar uma política ótima calculando diretamente valores para a função $Q(s, a)$ para todos os estados e ações de um MDP, busca aproximar, de maneira iterativa, os valores para a função $Q(s, a)$ para os estados e ações do sistema através de uma busca baseada em tentativa e erro, uma vez que o algoritmo não supõe que o agente possui quaisquer informações acerca das transições entre estados e das recompensas associadas a elas.

A ideia fundamental do algoritmo Q-Learning é, portanto, a aproximação dos valores para a função Q através dos valores $Q(s, a)$ que são observados na medida em que o agente interage com o ambiente. Tais valores (chamados de *Q-values*) são salvos em uma matriz de tamanho $|S| \times |A|$ (normalmente discretiza-se representações contínuas, como quando um estado é representado por um número real) chamada de *Q-table* e são sucessivamente atualizados para valores cada vez mais próximos dos valores verdadeiros da função $Q(s, a)$. Os valores da *Q-table* são atualizados com base na seguinte fórmula:

$$Q_{t+1}(s, a) = (1 - \alpha)Q_t(s, a) + \alpha(R(s, a) + \gamma \max_{a \in A} Q'(s', a)) \tag{2.5}$$

Onde α é chamado de *coeficiente de aprendizado*, que diz qual o peso do novo valor observado ante o *Q-value* guardado na tabela, oriundo de observações anteriores.

O algoritmo de Q-learning é definido pelo pseudocódigo presente no algoritmo 1.

Algorithm 1 Algoritmo de Q-learning

Entrada: Um MDP (S, A, T, R) e um coeficiente de aprendizado α
Saída: Uma função Q, aproximação de Q^*

1: **função** QLEARNING(MDP)
2: Inicialize $Q : S \times A \mapsto \mathbb{R}$ aleatoriamente
3: **repita**
4: Inicialize o agente em um estado inicial $s \in S$
5: **enquanto** *s não é um estado terminal* **faça**
6: Calcular a de acordo com a política atual de exploração (e.g. $\arg\max_a Q(s, a)$)
7: $s' \leftarrow T(s, a)$
8: $Q(s', a) \leftarrow (1 - \alpha) \cdot Q(s, a) + \alpha \cdot (r + \gamma \max_{a' \in A} Q(s', a'))$
9: $s \leftarrow s'$
10: **fim enquanto**
11: **até que** *critério de convergência de Q seja atingido*
12: **fim função**

2.4.1.1 Política de exploração em Q-Learning

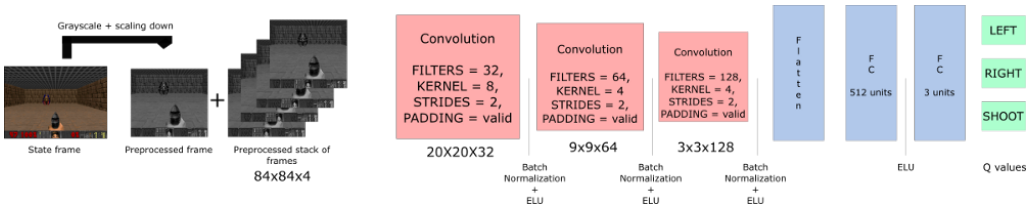
Na linha 6 do algoritmo 1, uma ação, dentre todas as possíveis ações para o estado no qual o agente se encontra, deve ser escolhida. Mas, em um estado inicial, apenas com valores gerados aleatoriamente para a *Q-table*, como o agente deve decidir qual a melhor ação a ser tomada? E mais: após algumas iterações, quando já se sabe que algumas sequências de ações geram as melhores recompensas (por trazerem na *Q-table* os melhores valores para as aproximações da função Q), como saber se outras sequências, ainda não exploradas, não resultam em recompensas totais maiores? A cada momento, portanto, o agente deve decidir se busca melhorar as sequências de ações já encontradas até o momento através do seu refinamento (o que é chamado de *exploitation*) ou se busca sequências de ações completamente novas, na esperança de que elas resultem em recompensas totais maiores (o que é chamado de *exploration*). O problema que trata dessa decisão a ser feita pelo agente é chamado de *exploration/exploitation trade-off* ou de *exploration/exploitation problem*. A solução para esse problema proposta em (WATTKINS, 1989) é simples: há um fator $\epsilon \in [0, 1]$ que determina a probabilidade de, a cada época de decisão, o agente não seguir os valores da *Q-table* e, ao invés disto, executar uma ação aleatória dentre todas as outras possíveis para aquele estado. No começo da interação

do agente com o ambiente, quando a maioria das células da *Q-table* ainda guardam os valores gerados aleatoriamente no primeiro passo do algoritmo, ϵ guarda um valor próximo de 1, fazendo com que o agente tome uma decisão aleatória (e, consequentemente, exploratória) na grande maioria das vezes. À medida que o agente atualiza os valores da *Q-table*, entretanto — reunindo, desta maneira, mais conhecimento sobre o sistema —, o valor de ϵ decai, fazendo com que o agente passe, gradativamente, a tomar mais decisões baseadas nos valores da *Q-table*.

2.4.2 Aprendizado por Reforço com Redes Neurais Profundas

O algoritmo chamado de *Deep Q-Learning* tem este nome porque combina *Q-Learning* com redes neurais profundas (*deep neural networks*). Seu funcionamento é parecido com *Q-Learning*, com a diferença de que, ao invés serem guardados em uma tabela, os valores aproximados para a função $Q^*(s, a)$ são fornecidos a uma rede neural profunda, que usa os valores dados a ela para aproximar a função Q^*. Possui a vantagem da escalabilidade, uma vez que sistemas onde os estados possuem muitas dimensões (ou há muitas ações possíveis para cada estado) requerem tabelas que, na prática, acabam se tornando inviáveis, no que tange ao consumo de recursos computacionais (e.g. memória). Em (MNIH et al., 2013), uma rede neural profunda foi treinada para jogar jogos da plataforma Atari: neste trabalho, o estado é representado pelos componentes visuais do jogo, ou seja, pela imagem formada na tela. Quatro quadros consecutivos do jogo são transmitidos a uma rede neural convolucional (LECUN et al., 1999), como registrado na figura 2.2, que tenta aproximar, para cada ação possível, seu valor Q, sendo tomada a ação de maior valor. O algoritmo aproveita-se do fato de que, em um jogo, há um número muito limitado de ações possíveis para cada estado: mover-se ou disparar uma arma por exemplo. Os treinamentos resultaram em performances superiores à humana em todos os jogos para os quais redes foram treinadas.

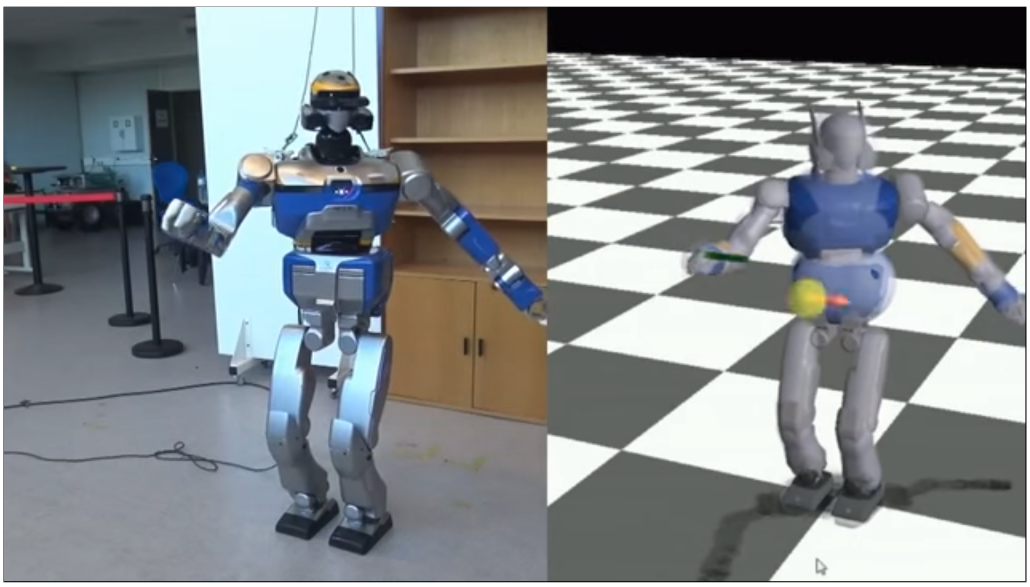
Figura 2.2: Esquema de uma rede neural para o jogo *Doom*.



Talvez o exemplo mais famoso de aplicação que combinou algoritmos clássicos de aprendizado por reforço com redes neurais profundas tenha sido o AlphaGo (SILVER et al., 2016). Desenvolvido pela empresa DeepMind, o AlphaGo é um *software* que joga o jogo de tabuleiro Go, original da China e talvez um dos jogos de tabuleiro mais antigos do mundo. Ensinar programas de computador a jogar Go nunca foi uma tarefa tão simples como jogar xadrez, por exemplo, uma vez que o número de jogadas possíveis em uma rodada é muito maior, tornando buscas que usam árvores para representar jogadas futuras praticamente inviáveis (BURMEISTER; WILES, 1995). Este problema, entretanto, foi contornado pelos pesquisadores da DeepMind através de um algoritmo que combina buscas guiadas por políticas com redes neurais profundas, e o resultado foi uma série de vitórias contra campeões continentais e mundiais da modalidade. Tais feitos não seriam possíveis, entretanto, se não houvesse alguma maneira de representar o jogo em um ambiente virtual, seja o jogo feito para a plataforma Atari ou o jogo de tabuleiro, e trabalhos como os citados aqui só são possíveis com o uso de simuladores.

2.5 Simuladores de AR

Figura 2.3: Um robô real e sua versão modelada em um simulador.



Conforme mencionado anteriormente, algoritmos de aprendizado por reforço possuem diversas aplicações; seja em jogos de tabuleiro ou de *video games* (CHEN, 2016; BELLEMARE et al., 2013; KEMPKA et al., 2016) ou sistema de controle de veícu-

los autônomos (NG, 2003), algoritmos de aprendizado por reforço são recomendados em quaisquer situações onde ações devem ser tomadas em sequência, dentro de um ambiente. Um problema surge, entretanto, quando torna-se necessário modelar um ambiente sob a forma de um Modelo de Markov ou sob qualquer outra forma que possa ser compreendida por um programa de computador. No caso do AlphaGo, o tabuleiro e as peças precisavam ser representados em um *software* para que o agente em treinamento pudesse lê-los e compreendê-los; no caso trabalhado por Andrew Ng (NG, 2003), uma representação do veículo automotor precisa ser modelada, para que o comportamento que mantém o veículo estabilizado possa ser aprendido com a ajuda dos métodos propostos pelo trabalho.

No caso da robótica, o uso de simuladores possui ainda outra vantagem prática: modelar robôs em um ambiente simulado que segue as leis da física do mundo real permite, por exemplo, que um modelo de um robô passe por um treinamento virtual antes de ser construído no mundo real (como na figura 2.3), acelerando, desta maneira, o processo de treinamento. Há diversas ferramentas que são utilizadas especialmente para construir simulações que apresentem características físicas semelhantes às do mundo real: motores de física como o MuJoCo (TODOROV; EREZ; TASSA, 2012), por exemplo, fornecem uma das bases para o *software* da DeepMind (TASSA et al., 2018), que pode ser utilizado para a modelagem de problemas de robótica.

2.5.1 Definição de um simulador

Em sua definição de dicionário, um simulador é "uma máquina com um conjunto de controles designada para proporcionar uma imitação realística da operação de um veículo, aeronave ou outro sistema complexo, usado para fins de treinamento". Um simulador, portanto, é um sistema capaz de criar um ambiente virtual cujas características aproximem-se, com certo grau de fidelidade, das condições apresentadas por um ambiente real. Tomemos as autoescolas do Brasil como exemplo: nelas, entre o fim das aulas teóricas e o começo das aulas práticas, dadas em um veículo real, simuladores (figura 2.4) são utilizados por alunos para que os princípios básicos de direção sejam aprendidos sem o risco de acidentes.

Obviamente, os simuladores de direção não são uma representação completamente fiel de um automóvel de verdade, mas são uma aproximação fiel o suficiente para que aquele que almeja adquirir sua carteira de habilitação possa embarcar em um automóvel sabendo ao menos o básico de como guiá-lo. Aqui, há uma espécie de troca entre fideli-

dade e segurança: obviamente os simuladores de direção não apresentam ao seu usuário todas as nuances de um automóvel e todas as particularidades do trânsito, afinal o simulador possui caráter introdutório e apenas uma parte das inúmeras situações que um motorista enfrenta é apresentada. Há a vantagem da segurança, entretanto: no momento em que o motorista em formação embarca em um veículo verdadeiro, os conhecimentos básicos necessários para dirigir já foram adquiridos, o que reduz o risco de acidentes causados por falta de preparo por parte do aluno. Além das questões relacionadas a segurança no trânsito, em um simulador de direção, situações corriqueiras do trânsito podem ser ensaiadas, sem que o aluno tenha que vivenciá-las na prática. Isto fornece aos preparadores um grau maior de controle sobre os diferentes desafios que são apresentados àqueles que pretendem obter sua licença para dirigir: cenários que ocorrem com pouca frequência, por exemplo, podem ser apresentados em um simulador repetidas vezes para o aluno, até que este adquira o domínio necessário para enfrentar a mesma situação na prática.

Figura 2.4: Simulador usado em autoescolas brasileiras.

Da mesma maneira, no trabalho apresentado em (NG, 2003), uma modelagem virtual das diferentes propriedades do helicóptero lidas pelo *software* de controle é construída. Neste ambiente simulado, alimentado com dados resultantes de um experimento onde o helicóptero é controlado por um piloto humano, o algoritmo proposto pelo trabalho tenta aproximar-se do comportamento do piloto, aprendendo a guiar o veículo sem grandes perturbações. Neste caso, fica muito mais evidente o grau de controle que o simulador

fornece aos envolvidos no projeto de desenvolvimento do veículo: nele, os dados gerados a partir de um piloto humano podem ser manipulados para forçar o agente a aprender a comportar-se em diversos tipos de situação. Isto acarreta em uma enorme economia de tempo e também de recursos, uma vez que, quando um acidente destrutivo ocorre dentro de um ambiente simulado, não é necessário construir uma nave novamente.

Com base nos exemplos dados acima, é fácil inferir que há vantagens associadas ao uso de simuladores. O que não fica tão evidente, entretanto, é o custo que a adoção de tais ferramentas traz consigo: além, obviamente, dos recursos que devem ser empregados para que se tenha um simulador pronto para uso — aqui, podem ser considerados recursos o tempo necessário para desenvolver um simulador quanto o dinheiro para pagar pela licença de um *software* já pronto —, há também o custo relativo ao tempo que é necessário para que sejam dominadas todas as ferramentas que um simulador oferece. Tais custos, entretanto, trazem consigo as vantagens observáveis nos dois exemplos apresentados: no caso da autoescola, o risco de acidentes é reduzido, uma vez que os condutores em formação já possuem os conhecimentos básicos de direção ao conduzir um veículo pela primeira vez; no caso do *drone*, por sua vez, o tempo de treinamento do veículo é reduzido drasticamente, acarretanto em uma economia de tempo.

O uso de simuladores, assim sendo, traz consigo um custo normalmente associado a recursos financeiros ou tempo, que precisam ser empregados para que o *software* seja adquirido ou para que as ferramentas fornecidas por ele sejam compreendidas e tenham seu uso dominado. Tais custos, entretanto, podem ser compensados — ou até mesmo amplamente superados — pelas diferentes vantagens trazidas pelo uso de tais ferramentas. Cabe, neste caso, uma análise minuciosa das vantagens e desvantagens da adoção de simuladores, bem como uma escolha sensata das diferentes ferramentas disponíveis.

2.6 Simuladores de Aprendizado por Reforço

Algoritmos de aprendizado por reforço são recomendados em quaisquer situações onde ações devem ser tomadas de maneira sequencial por um agente inserido dentro de um ambiente, com o qual ele interage através da sua sucessão de decisões tomadas. Problemas diferentes, portanto, requerem diferentes meios de simulação: um agente que joga xadrez (Silver et al., 2017) pode ser treinado usando-se de um simulador que representa o tabuleiro através de uma matriz, onde cada célula guarda a informação da peça ali presente, caso haja alguma. Um outro simulador (TOGELIUS et al., 2009), que treina um

agente para jogar o jogo *Super Mario World*, baseia-se em uma adaptação do modo com que os diferentes níveis do jogo eram representados no *Super Nintendo*, seu *console* de origem.

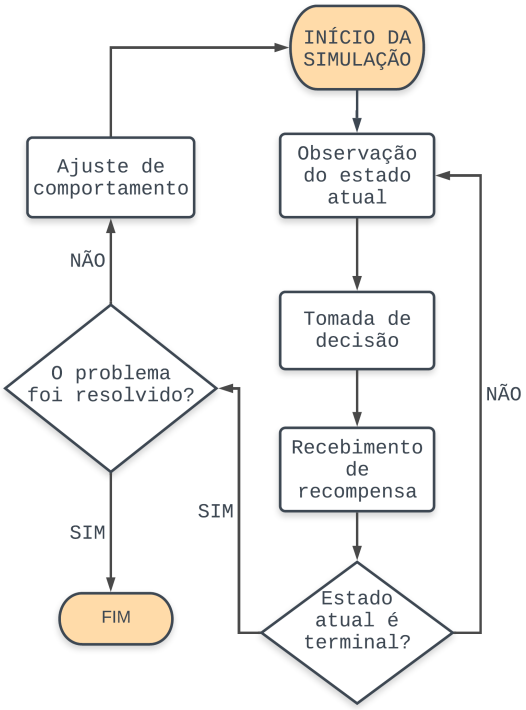
Visto que problemas diferentes requerem maneiras distintas de representar-se o ambiente no qual os agentes estão inseridos e com os quais eles interagem, é necessário definir a estrutura básica, comum a qualquer simulação, de um problema de aprendizado por reforço. É natural que, ao pensarmos na palavra "simulador", nos venha à mente algum tipo de ferramenta essencialmente visual, como o simulador utilizado em autoescolas da figura 2.4. Entretanto, quando se trata de aprendizado por reforço, nem sempre há utilidade em construir visualmente o ambiente e o agente nele inserido: como no caso do tabuleiro de xadrez, representá-lo através de figuras visuais é interessante, mas opcional.

Tratanto-se de aprendizado por reforço, portanto, o componente visual nem sempre faz parte da essência do simulador. Por vezes, é util que este seja ignorado para que os recursos que seriam consumidos por algoritmos de computação gráfica sejam transferidos às tarefas ligadas ao treinamento do agente e, em alguns casos, é até mesmo impossível representar o problema de uma forma essencialmente visual (NARASIMHAN; KULKARNI; BARZILAY, 2015). Sendo assim, resta definir qual é o elemento fundamental de um simulador, quando dentro de um contexto de treinamento de agentes através de algoritmos de aprendizado por reforço.

E qual é a peça principal de um sistema modelado por um Processo de Decisão de Markov? Bom, é natural que seja, exatamente, o MDP e seus componentes: as informações a respeito do ambiente, as ações possíveis que podem ser tomadas em um estado, a noção de recompensa e as transições entre estados. O componente fundamental de uma simulação de aprendizado por reforço é, portanto, a representação dos elementos que compõem um Processo de Decisão de Markov e das relações que estes elementos possuem entre si, como as transições entre estados decorrentes das ações tomadas pelo agente e as recompensas recebidas por elas. A figura 2.5 mostra o fluxo percorrido durante o treinamento de um agente de aprendizado por reforço, e é justamente a implementação dos estágios deste fluxo que compõem o sistema fundamental de treinamento de um agente de AR.

Um simulador de aprendizado, portanto, combina o ciclo descrito na figura 2.5 com ferramentas que permitem que sejam representados os componentes de um MDP: o ambiente, o agente nele inserido, as ações possíveis em cada estado e a noção de recompensa, sem que, necessariamente, haja uma representação visual destes. A ferramenta

Figura 2.5: Ciclo de aprendizado do agente via interação com o ambiente



VizDoom (KEMPKA et al., 2016), por exemplo, permite que algoritmos de aprendizado por reforço sejam aplicados ao jogo de tiro em terceira pessoa *Doom*, tendo se tornado uma ferramenta importante de avaliação de algoritmos de AR. Há evidentemente, uma construção visual dos objetos (i.e. o jogador, o mapa e os inimigos), mas também há, no cerne do *software*, estruturas de dados internas que representam o estado atual do jogo e das quais o agente extrai as informações necessárias para suas tomadas de decisão. Acima destas estruturas, o treinamento do agente é realizado seguindo-se o fluxo da figura 2.5.

2.6.1 Estrutura simulador de aprendizado por reforço

Figura 2.6: O problema conhecido como *Frozen Lake*

Para construir um simulador de AR, conforme desenvolvido anteriormente, não é estritamente necessária a presença de um componente visual. É preciso, entretanto, que a arquitetura do simulador contenha os componentes de um MDP e que as relações entre eles se deem corretamente, em um ciclo próximo do que é descrito na imagem 2.5. Estas são as premissas básicas para a construção de um simulador de aprendizado por reforço e qualquer construção que vá além desta definição irá depender da natureza do problema a ser resolvido, o que pode exigir o uso de ferramentas mais complexas, como motores de física, que buscam aproximar o comportamento de corpos no espaço tridimensional da maneira como ocorre no mundo real, ou o emprego de ferramentas mais simples. Um exemplo de problema bastante trivial, que pode ser construído em um simulador sem que seja necessário nada além de estruturas básicas de dados e representação visual em modo texto é um problema chamado *Frozen Lake* (OPENAI, 2019): o problema descreve um homem que precisa andar por sobre um lago congelado, saindo de um ponto de partida e com destino a um ponto de chegada, previamente conhecido. O lago apresenta lugares, porém, onde a camada de gelo é fina demais para suportar o peso do homem, que cairá na água caso tente passar por eles. Cabe ao agente, portanto, aprender a controlar o homem por sobre o lago, sabendo de onde ele sai, para onde ele deve se dirigir e quais pontos no lago não podem ser perpassados. Para representar o lago, apenas uma matriz de caracteres é necessária, com símbolos distintos para representar o ponto de partida, o ponto de chegada, os pontos onde é seguro atravessar e os pontos onde o gelo é fino demais. Uma matriz de tamanho arbitrário, descrita na tela de um *console*, portanto, é capaz de fornecer uma representação completa do problema, como é o caso na imagem 2.6.

Cabe esclarecer, entretanto, que o caso do problema *Frozen Lake* é bastante especial, uma vez que o problema é comumente utilizado apenas para fins didáticos, como exemplo de uso de técnicas simples de resolução de Processos de Markov como as exibidas no capítulo 2. É natural que o uso de simuladores mais complexos, que tenham a capacidade de representar estados definidos por um volume maior de informações, sejam empregados quando o problema em questão envolve mais elementos do que uma simples matriz que representa um lago congelado.

É este o caso no ramo da robótica, por exemplo. Qualquer tarefa que necessite ser executada repetidamente e com certo grau de precisão é uma boa candidata para a automatização através do emprego de robôs: linhas de montagem de empresas automotivas, por exemplo, já são dominadas por robôs há varios anos. Outras tarefas, mais complexas

do que as executadas pelos robôs da manufatura, mas ainda assim consideradas maçantes e repetitivas por humanos — como dirigir, por exemplo —, também são fortes candidatas à automação, tendo, nos últimos tempos, sido alvo de pesquisas de ponta envolvendo inteligência artificial, com a popularização de técnicas de aprendizado de máquina e visão computacional. É aí que entra o uso de simuladores: durante as fases iniciais de desenvolvimento de um robô, uma versão virtual deste pode ser construída, como na figura 2.3, em um ambiente fundamentado em motores de física que simulam as leis do mundo real. No ambiente controlado de um simulador, o treinamento do agente pode ser feito de forma acelerada, fazendo com que as primeiras versões reais do robô já apresentem uma versão incial do comportamento que quer-se ensinar a ele.

No capítulo a seguir, serão discutidas as diferentes formas de implementar-se uma simulação de um problema de aprendizado por reforço, bem como as ferramentas que podem ser utilizadas para este fim. Também é apresentada uma hierarquia, mostrando o grau de complexidade de cada uma das ferramentas.

3 ESTADO DA ARTE

No capítulo anterior, foi apresentada a formulação matemática de um problema de AR e discutiu-se a respeito de simuladores de aprendizado por reforço, onde discorreu-se sobre as vantagens de usar simuladores em problemas práticos bem como na estrutura comum que é compartilhada por todos os simuladores de AR. Neste capítulo, serão apresentadas diferentes maneiras de se construir um simulador que representa um problema a ser resolvido por um agente de AR. Alguns recursos que podem ajudar no processo, como motores de física, serão apresentadas e, ao final do capítulo, será proposta uma hierarquia na qual as diferentes ferramentas se organizam de acordo com o seu grau de especificidade.

3.1 Construindo um simulador de aprendizado do zero

Como qualquer outro problema do universo da ciência da computação, é possível construir um simulador de aprendizado por reforço a partir dos elementos básicos de entrada, processamento e saída de dados fornecidos por linguagens de programação como *Python* e C++. Tomemos como exemplo um problema onde um agente é responsável por copiar uma sequência genética. No problema, para uma sequência de n genes, o agente recebe 1 como recompensa para um gene copiado corretamente e 0 para genes copiados incorretamente. Construir uma simulação deste problema é bastante simples: basta que seja usada uma sequência de caracteres para representar os genes. O código referente ao problema está listado no código-fonte 3.1. No código, é possível perceber que nenhuma ferramenta além daquelas que são oferecidas pelo pacote básico da linguagem *Python* é utilizada: um gerador de números inteiros, oferecido nativamente pela linguagem, é o mecanismo mais complexo colocado em uso. A estrutura de aprendizado por reforço e o ciclo da figura 2.5 também estão bastante claros no código.

```
1 from random import randint
2
3 MIN_SIZE = 10
4 MAX_SIZE = 100
5 EPISODES = 2
6 GENES = ['G', 'C', 'A', 'T']
```

```
7
8  # definição do problema
9  class DNACopy():
10     def __init__(self):
11         self.init()
12
13     # inicialização
14     def init(self):
15         self.i = 0
16         self.sequence = []
17         self.copy = []
18         size =
19         for i in range(randint(MIN_SIZE, MAX_SIZE)):
20             self.sequence.append(GENES[randint(0, 3)])
21         return self.sequence[i]
22     def size(self):
23         return len(self.sequence)
24
25     # método de tomada de ação
26     def action(self, g):
27         self.copy.append(GENES[g])
28         self.i += 1
29         return self.reward(g)
30
31     # método de observação
32     def observation(self):
33         return self.sequence[self.i]
34
35     # método de cálculo de recompensa
36     def reward(self, g):
37         return int(self.sequence[self.i] == \
38          GENES[g])
39
40 for e in range(EPISODES):
```

```
41      problem = DNACopy()
42
43      # inicialização do problema e da recompensa
44      problem.init()
45      reward = 0
46
47      # ciclo de tomada de ações
48      for action in range(problem.size() - 1):
49          # ação aleatória é tomada e
50          # a recompensa é somada à variavel reward
51          a = random.randint(0, len(GENES) - 1)
52          reward += problem.action(a)
53      print("Total reward is %d" % reward)
```

Código-fonte 3.1: Exemplo de um problema de cópia de material genético.

O problema é bastante simples: para um material genético de tamanho m, a recompensa total recebida é n/m, onde n é o número de genes copiados corretamente. Mas e para problemas mais complexos, que envolvem não sequências de caracteres que representam genes, mas objetos localizados em um espaço de duas ou três dimensões, como representar os elementos do problema nativamente? Problemas mais complexos requerem, naturalmente, o uso de ferramentas mais complexas para sua construção e motores de física, ferramentas capazes de representar objetos no espaço e de simular as o contato entre eles, são um belo exemplo disso. A construção de motores de física que executam cálculos que simulam a interação entre objetos em espaços multidimensionais faz parte de um amplo campo de pesquisa e desenvolvimento, então é natural que seja desnecessário para quem está desenvolvendo um simulador de aprendizado por reforço implementar um novo motor do zero.

3.2 Motores de Física

Um motor de física nada mais é do que um programa que simula leis de Newton utilizando princípios como massa, velocidade, atrito e resistência do ar, tornando possível a modelagem de situações que envolvem a física do mundo real. Muito utilizados nos campos de pesquisa e de jogos eletrônicos, motores de física são essenciais para garantir a verossimilhança de uma simulação de um problema de robótica ou de um jogo, por

exemplo.

Robôs executam tarefas através do contato físico com o ambiente que os cerca. É primordial, portanto, para o desenvolvimento do *software* que compõe o robô, que existam ferramentas capazes de reconstruir, com elevado grau de precisão, em um ambiente virtual e controlado, os aspectos físicos de todos os elementos que podem estar à sua volta. Para isso, um simulador deve definir os corpos no espaço e calcular as forças resultantes do contato entre eles, o que é um problema NP-difícil (KAUFMAN et al., 2008). Para tratar o problema, motores de física utilizam de versões relaxadas do problema, permitindo que este seja tratado por métodos de solução que envolvem programação linear.

A ideia por trás de um motor de física é simples: define-se um intervalo de tempo δ e, para um instante de tempo t, todas as interações entre os objetos representados e as resultantes das forças neles aplicadas são calculadas, e o resultado destes cálculos são as posições dos objetos e as forças que sobre eles atuam no instante de tempo $t + \delta$. Imprecisões são proporcionais a δ, ou seja, um simulador de física só é completamente preciso se o intervalo δ é infinitamente pequeno.

Em um mundo ideal, simulações feitas por um motor de física teriam ao mesmo tempo um custo computacional pequeno e uma precisão próxima da realidade. O que acontece, entretanto, é uma espécie de *trade-off* entre a precisão da simulação e o tempo necessário para sua execução: em aplicações onde o mais importante é a velocidade da simulação, como em jogos eletrônicos — onde o δ é normalmente $\frac{1}{60}s$ — o sistema pode abrir mão da precisão. Em simulações utilizadas na robótica, por outro lado, a precisão é muito mais importante, uma vez que a simulação pode anteceder um experimento feito no mundo real e imprecisões no motor de física podem levar a resultados completamente divergentes da realidade. Conforme apontado em (SIMS, 1994), imprecisões em simulações serão invariavelmente exploradas por algoritmos de otimização.

Todas estas nuances devem ser levadas em consideração na hora de se escolher ou desenvolver um motor de física, o que torna complicada a tarefa de decidir qual será a ferramenta utilizada para a simulação que se deseja desenvolver. Desenvolver um motor de física próprio, entretanto, é uma tarefa ainda mais complexa. O mero cálculo de forças que atuam sobre corpos unidos por juntas, por exemplo, pode requerer o uso de diversas fórmulas (SMITH, 2002). Para aqueles que desejam desenvolver aplicações de aprendizado por reforço, portanto, surge a necessidade de se estudar diferentes ferramentas de física, o que pode consumir bastante tempo.

Há vários motores de física disponíveis, alguns com custo e outros sob licenças de

código aberto. Os mais frequentemente utilizados, bem como exemplos de aplicações de aprendizado por reforço onde eles são empregados, são apresentados a seguir. Alguns dos motores apresentados fornecem simulações robustas, porém mais lentas; outros, por sua vez, por serem voltados à aplicações visuais, realizam as simulações mais rapidamente, mas com perda de precisão. Foram trazidos para este trabalho três motores de física bastante conhecidos, sendo dois deles tridimensionais e um bidimensional. Além disso, este trabalho descreve também um estudo (KANG; HWANGHO, 2018), que compara não só os dois motores de física tridimensionais apresentados nesta seção, como dois outros simuladores de propósito mais específico, apresentados na seção seguinte.

O trabalho desenvolvido em (KANG; HWANGHO, 2018) apresenta um estudo comparativo entre diversos motores de física onde cada um dos cenários de teste verifica o comportamento de uma característica diferente das simulações. São sete cenários de teste ao todo:

- *Rolling test*: teste de modelos de fricção. Testa a capacidade dos motores de simular forças de atrito corretamente;
- *Bouncing test*: nesta simulação um objeto esférico cai no chão. Testa a capacidade dos motores de simular forças de colisão elástica;
- *666 balls test*: uma perturbação é inserida em uma pilha de tamanho $6 \times 6 \times 6$ de objetos esféricos. Testa a capacidade dos motores de lidar com colisões sólidas envolvendo múltiplos corpos;
- *Elastic 666 balls test*: mesma situação que o cenário anterior, porém com colisões elásticas;
- *ANYmal PD control test*: neste cenário, um robô quadrúpede (HUTTER et al., 2016) deve permanecer em pé. O cenário testa a precisão da representação das juntas e a escalabilidade do motor, uma vez que o teste é repetido várias vezes para um número crescente de robôs;
- *ANYmal momentum test*: neste cenário, um objeto esférico colide com o robô. É testada a capacidade do motor de representar as juntas corretamente durante e após o momento da colisão;
- *ANYmal energy test*: neste cenário, o robô é objeto de uma força que o impulsiona para cima. O cenário testa a capacidade do motor de representar as juntas do robô corretamente durante a queda e no instante que ele colide com o chão.

3.2.0.1 Open Dynamics Engine

Sendo uma ferramenta de simulações físicas de propósitos gerais, a *Open Dynamics Engine* (SMITH, 2006) foi feita para simular propriedades físicas de corpos articulados. Este motor é ideal para simulações que possam envolver veículos ou criaturas bípedes, por exemplo. Com uma extensa biblioteca capas de gerar representações fidedignas dos mais variados corpos tridimensionais e de junções que os conectam, a ODE vem sendo mantida por mais de uma década, contando com uma comunidade bastante ativa de desenvolvedores. Entretanto, justamente por ser um motor de física relativamente antigo, a ODE acaba por não se beneficiar de tecnologias mais recentes, como paralelismo, por exemplo. Há registros de tentativas de expansão da ferramenta para adicionar o suporte a processamento paralelo, mas a tarefa se provou desafiadora (GOODSTEIN; ASHLEY-ROLLMAN; ZAGIEBOYLO, 2007).

Na avaliação conduzida em (KANG; HWANGHO, 2018) a *ODE* se destaca no cenário *ANYmal momentum test*, saindo-se ligeiramente melhor, no que se refere à acurácia da simulação (neste cenário, o erro é representado pela diferença quadrática entre a quantidade de energia que deveria haver no sistema em uma simulação ideal e a quantidade de energia presente no sistema simulado, medido em $(N \cdot m/s)^2$). Deve ser observado, entretanto, que, mesmo no cenário de teste em que a *Open Dynamics Engine* apresentou simulações de qualidade, o tempo utilizado pelo motor para computar as simulações é muito maior do que o tempo usado nos outros motores, o que se deve pelo fato da *ODE* ser um sistema antigo, e que portanto não se beneficia de ferramentas de paralelização fornecidas por processadores recentes.

Ainda assim, a *ODE* é amplamente utilizada em jogos, talvez pelo fato de ser um sistema de código aberto, licenciado sob a BSD e a LGPL, e também em razão do sistema ter sido portado para diversas linguagens de programação diferentes. Abaixo, um exemplo de código que utiliza a versão em *Python* da *ODE* para, juntamente com a biblioteca *Pygame*, desenhar um pêndulo.

```
1 # pyODE example 2: Connecting bodies with joints
2
3 import pygame
4 from pygame.locals import *
5 import ode
6
```

```
7
8  def coord(x,y):
9      "Convert world coordinates to pixel coordinates."
10     return 320+170*x, 400-170*y
11
12
13 # Initialize pygame
14 pygame.init()
15
16 # Open a display
17 srf = pygame.display.set_mode((640,480))
18
19 # Create a world object
20 world = ode.World()
21 world.setGravity((0,-9.81,0))
22
23 # Create two bodies
24 body1 = ode.Body(world)
25 M = ode.Mass()
26 M.setSphere(2500, 0.05)
27 body1.setMass(M)
28 body1.setPosition((1,2,0))
29
30 body2 = ode.Body(world)
31 M = ode.Mass()
32 M.setSphere(2500, 0.05)
33 body2.setMass(M)
34 body2.setPosition((2,2,0))
35
36 # Connect body1 with the static environment
37 j1 = ode.BallJoint(world)
38 j1.attach(body1, ode.environment)
39 j1.setAnchor( (0,2,0) )
40
```

```
41 # Connect body2 with body1
42 j2 = ode.BallJoint(world)
43 j2.attach(body1, body2)
44 j2.setAnchor( (1,2,0) )
45
46
47 # Simulation loop...
48 fps = 50
49 dt = 1.0/fps
50 loopFlag = True
51 clk = pygame.time.Clock()
52
53 while loopFlag:
54     events = pygame.event.get()
55     for e in events:
56         if e.type==QUIT:
57             loopFlag=False
58         if e.type==KEYDOWN:
59             loopFlag=False
60
61     # Clear the screen
62     srf.fill((255,255,255))
63
64     # Draw the two bodies
65     x1,y1,z1 = body1.getPosition()
66     x2,y2,z2 = body2.getPosition()
67     pygame.draw.circle(srf, (55,0,200),\
68       coord(x1,y1), 20, 0)
69     pygame.draw.line(srf, (55,0,200),\
70       coord(0,2), coord(x1,y1), 2)
71     pygame.draw.circle(srf, (55,0,200),\
72       coord(x2,y2), 20, 0)
73     pygame.draw.line(srf, (55,0,200),\
74       coord(x1,y1), coord(x2,y2), 2)
```

```
75
76      pygame.display.flip()
77
78      # Next simulation step
79      world.step(dt)
80
81      # Try to keep the specified framerate
82      clk.tick(fps)
```

Código-fonte 3.2: Exemplo de uma simulação que usa a Open Dynamics Engine.

É necessário observar no código acima que as chamadas da função que atualiza as propriedades físicas de cada objeto depois de um intervalo dt e da função que desenha os objetos criados não é automática, sendo de responsabilidade do programador. No capítulo seguinte deste trabalho, será exposto como a ferramenta proposta tenta simplificar estas tarefas.

3.2.0.2 Bullet Physics

O motor de física *Bullet Physics* (COUMANS, 2013) tinha como propósito principal ser usado para jogos eletrônicos, tendo também sido usado na criação de efeitos especiais em filmes. Quanto à sua performance, a ferramenta apresenta um comportamento parecido com a *ODE*, com a diferença de que a *Bullet* apresenta um grau de sofisticação maior em simulações que devem levar forças de fricção em consideração, executando as simulações relacionadas a esta força de maneira eficiente e acurada e, portanto, apresentando um desempenho melhor nos cenários relacionados em (KANG; HWANGHO, 2018). Assim como a *ODE*, a biblioteca *Bullet* apresenta um desempenho melhor quando o objeto é representado por apenas um corpo, formado por uma das primitivas (i.e. unidades básicas de construção) da ferramenta, enquanto o *MuJoCo*, por exemplo, se comporta melhor quando a simulação envolve corpos complexos.

Dois trabalhos construídos com a ajuda da *Bullet Physics* merecem atenção: em um deles, um robô formado por um braço e um par de pinças, com uma caixa cheia de objetos disposta à sua frente, deve inferir, a partir de informações visuais, como pegar objetos de dentro da caixa e arremessá-los em uma outra caixa, disposta um pouco mais distante (figura 3.1). O treinamento do robô, chamado de *TossingBot* (ZENG et al., 2019), envolveu uma simulação utilizando-se da *Bullet Physics*. É importante notar a natureza da simulação, que envolveu uma série de corpos (no caso, os objetos da caixa) sem nenhum

Figura 3.1: TossingBot

tipo de conexão entre si, cenário para o qual este motor de física é recomendado. Em outro trabalho (PENG et al., 2018), um agente deve analisar imagens de atores, captadas por câmeras ou por ferramentas de *motion capture*, e fazer com que um robô, simulado em um ambiente que usa a biblioteca *Bullet*, imite os movimentos ou desenvolva variações deles.

Bullet apresentou boa performance nos cenários de teste apresentados em (KANG; HWANGHO, 2018) e, além disso, conta com diversas bibliotecas que a integram a várias linguagens de programação, como *Python*, por exemplo. Esta combinação de praticidade e boa performance é, provavelmente, um dos motivos que fez com que os desenvolvedores do simulador *Gazebo* tenham adotado *Bullet* como um dos motores de física que compõem a base das simulações da ferramenta.

3.2.0.3 Box2D

Há situações em que um motor de física tridimensional é desnecessário. Em jogos eletrônicos, por exemplo, quando, por uma opção de *design*, o jogo é desenvolvido em um ambiente bidimensional, não há a necessidade de usar motores que usem mais do que duas dimensões para representar seus objetos. O grau de complexidade de motores bidimensionais, entretanto, é praticamente o mesmo de motores de três dimensões, uma vez que o problema de simular interações físicas em ambientes bidimensionais também deve ser tratado de maneira aproximada, fazendo com que motores do tipo apresentem o

mesmo *trade-off* entre custo computacional e acurácia.

De todos os motores bidimensionais, talvez o mais conhecido seja o *Box2D* (CATTO, 2007). Criado para o desenvolvimento de jogos, a ferramenta possui versões para diversas linguagens de programação diferentes. O trecho de código 3.3 visa evidenciar as semelhanças entre um programa que utiliza Box2D e o código-fonte 3.2, que utiliza Open Dynamics Engine. Em ambos, o a função que atualiza as propriedades físicas de cada objeto deve ser chamada e um instante de tempo deve ser fornecido.

```
1 for (int i = 0; i < 60; ++i)
2 {
3  world.Step(timeStep, velocityIterations, positionIterations);
4  b2Vec2 position = body->GetPosition();
5  float32 angle = body->GetAngle();
6  printf("\%4.2f \%4.2f \%4.2f\n", position.x, position.y, angle);
7 }
```

Código-fonte 3.3: Trecho de um programa escrito em C++ que utiliza Box2D

Fica evidente que, mesmo se tratando de um ambiente bidimensional, o motor de física possui um certo grau de sofisticação e que permite uma ampla gama de experimentos. A grande maioria dos ambientes do Gym, por exemplo, é representada utilizando-se de ambientes de duas dimensões. Além disso, alguns problemas clássicos da literatura, como o *CartPole* (BARTO; SUTTON; ANDERSON, 1983), podem ser modelados em sistemas 2-D. Em vista disso, o Box2D foi o motor de física escolhido para compor este trabalho — uma descrição detalhada das ferramentas do Box2D utilizadas por este trabalho pode ser encontrada no capítulo 4.

Um exemplo de pesquisa que utiliza Box2D para a reprodução de seus resultados pode ser encontrada em (GEBHARDT et al., 2017). Neste trabalho, um enxame de robôs, modelados no ambiente da ferramenta, deve aprender a resolver tarefas de maneira coletiva.

3.3 Simuladores de propósito específico

Além de motores de física, desenvolvidos com propósitos gerais, há um outro tipo de ferramenta de simulação, de propósito mais específico. Normalmente, elas apresentam escopo reduzido em relação a ferramentas de propósito mais amplo, mas, por outro lado,

fornecem atalhos para o desenvolvimento de simulações que não fogem à sua proposta. Duas delas, DART e MuJoCo, voltadas para a área da robótica, são estudadas a seguir.

3.3.1 DART

Voltado, assim como o *MuJoCo*, para simulações envolvendo robótica, a ferramenta chamada *DART* (de *Dynamic Animation and Robotics Toolkit*) foi desenvolvida em 2012. Nos testes apresentados em (KANG; HWANGHO, 2018), apresentou resultados ruins, se comparados aos demais motores de física testados.

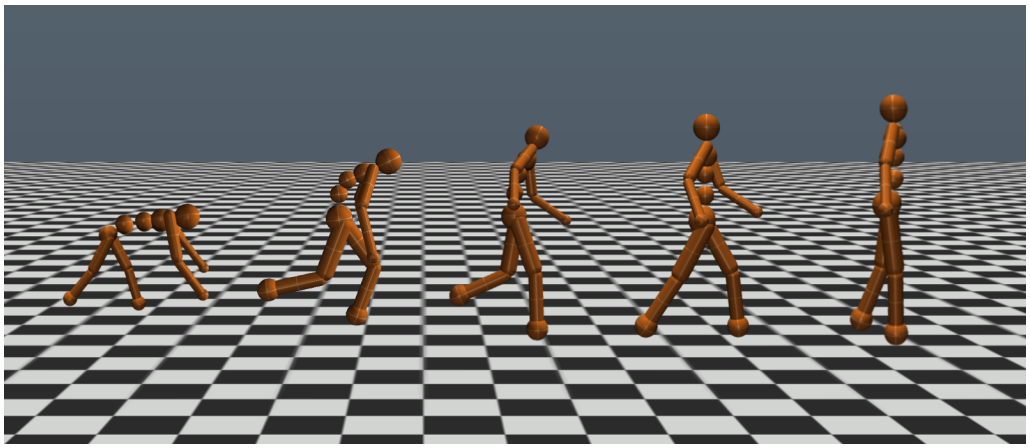
DART apresenta um comportamento diametralmente oposto ao motor de física ideal: os erros nos cenários de teste são consideráveis, ao mesmo tempo em que o custo computacional das simulações também é alto. Apesar de não possuir um desempenho excepcional, entretanto, *DART* foi adotada como um dos motores de física que formam a base do simulador *Gazebo* (KOENIG; HOWARD, 2004), voltado especialmente para a robótica e que possui uma linguagem de definição própria para representar modelos de robôs. O simulador é capaz de importar arquivos que, utilizando uma sintaxe própria, definem robôs e permite criá-los no ambiente simulado de uma maneira relativamente simples.

O fato do simulador *Gazebo* ser amplamente adotado na comunidade de pesquisa mesmo tendo como base um simulador de física que apresenta imprecisões consideráveis em suas simulações e que possui um custo computacional relativamente alto indica que, mesmo com um motor de física que está longe do ideal, ferramentas que simplificam a modelagem de cenários, como a linguagem própria do *Gazebo*, podem conferir um diferencial a um simulador.

3.3.2 MuJoCo

MuJoCo (TODOROV; EREZ; TASSA, 2012) é um *framework* surgido do rápido crescimento do ramo da robótica e da necessidade dos simuladores utilizados nesta área de acompanhar o acentuado desenvolvimento — não apenas em termos de capacidade, como também tratando-se de complexidade — deste domínio. O MuJoCo é voltado para o desenvolvimento de sistemas de controle para robôs, área fundamental da robótica que visa a automatização de tarefas antes realizadas por humanos ou por outros sistemas mais

Figura 3.2: Modelagem de robôs humanoides no ambiente do MuJoCo



simples.

O MuJoCo é voltado, portanto, para otimização de sistemas de controle, o que consiste, basicamente, em construir diversas variações de um sistema e avaliá-las de acordo com um processo que pode ser baseado em inteligência artificial, com o uso de algoritmos genéticos ou de aprendizado por reforço, por exemplo. Apesar, entretanto, de ter por objetivo apresentar uma solução para sistemas de robótica, onde é interessante representar virtualmente sistemas compostos de diversas partes conectadas de modo relativamente complexo (como é o caso de braços robóticos) e de possuir uma performance significativamente melhor do que outras ferramentas da área (no que tange a acurácia da simulação), o que acarreta em um custo computacional maior, e de possuir certas deficiências no tocante à simulações que não são normalmente empregadas em problemas de robótica (MuJoCo é incapaz de reproduzir colisões elásticas, por exemplo), MuJoCo foi rapidamente adotado por pesquisadores da área do aprendizado por reforço profundo (abordada brevemente na seção 2.4.2), onde algoritmos de aprendizado por reforço são combinados com redes neurais profundas.

Isto deve-se à facilidade com que modelos podem ser construídos no ambiente virtual do MuJoCo, o que pode ser feito através de uma descrição de alto nível com base em um arquivo no formato XML ou de chamadas diretas a uma API que fornece ao desenolvedor as ferramentas necessárias para a definição de modelos. Dado o modo simples através do qual objetos podem ser construídos em seu ambiente, é natural que este *framework* tenha se tornado uma peça importante no estudo de algoritmos de inteligência artificial quando aplicados em simulações, mesmo que sua performance em alguns dos cenários apresentados em (KANG; HWANGHO, 2018) seja baixa. Em especial, pode ser

citado o caso da DeepMind (HEESS et al., 2017), que desenvolveu uma extensa pesquisa envolvendo as diferentes maneiras com as quais robôs humanóides (figura 3.2) aprendem a se locomover de acordo com a maneira através da qual a noção de recompensa é representada.

Devido, entretanto, ao seu alto grau de especificidade (o MuJoCo foi feito especialmente para atender às demandas do ramo da robótica, onde motores de simulações físicas voltados para jogos não são acurados o suficiente — há casos, por exemplo, onde a simulação aprende a explorar falhas do motor de física, que não condizem com a realidade (SIMS, 1994)), há casos onde o *framework* pode ser considerado complexo demais, não só pela sua extensa documentação sobre as funções da sua API para criação de corpos e conectores, como também pelo seu conjunto de algoritmos de cálculos físicos, que fornecem uma lista bastante abrangente de informações sobre os corpos representados na simulação. Além do mais, há a questão do custo: o MuJoCo não é um *software* livre, sendo oferecido de graça apenas para integrantes da comunidade científica. Para os demais, sua licença é oferecida a preços que variam de quinhentos a dois mil dólares.

3.4 Simuladores com interface para Aprendizado de máquina

Apesar de executarem as tarefas para as quais foram propostos com bastante eficiência, os motores de física, sozinhos, não possuem a capacidade de construir uma simulação voltada para aprendizado por reforço, pois é preciso mais do que técnicas de resolução de forças vetoriais para dar inteligência a um agente. Um simulador de aprendizado por reforço precisa não só de um ambiente virtual (tal como os oferecidos pelos motores de física, mas sem que os ambientes possíveis estejam restritos aos fornecidos por eles), mas também de estruturas que representem as informações que representam os elementos envolvidos no ciclo de aprendizado de um agente de AR, como descrito na figura 2.5.

Felizmente, dependendo da natureza do agente e do problema que ele tenta resolver, não é necessário desenvolver um simulador de aprendizado por reforço completamente do zero. Há diversos *softwares* disponíveis no mercado que fornecem as ferramentas necessárias para treinar um agente sem que seja necessário preocupar-se com os elementos básicos envolvidos no treinamento, ficando a encargo do desenvolvedor somente a formulação do problema e do método de aprendizado.

Há uma série de problemas, de naturezas muito distintas, que técnicas de aprendi-

zado por reforço se propõem a resolver. É bastante diverso, justamente por isso, o universo de simuladores de agentes de AR. O desenvolvimento de um trabalho na área envolve, portanto, uma análise cuidadosa das ferramentas que serão utilizadas para a construção da simulação do problema: um *software* de propósito mais geral, por exemplo, pode ser utilizado sem problema algum para formular e resolver um problema do campo da robótica. Um outro *software*, voltado apenas para formulação de problemas desta área, entretanto, pode ser uma escolha melhor neste caso, uma vez que seu objetivo é justamente simplificar tarefas deste tipo.

Dois *softwares* para a formulação de problemas e treinamento de agentes de AR são apresentados a seguir. Um deles é voltado para a construção de cenários típicos de jogos eletrônicos ou de tabuleiro, enquanto o terceiro, sendo um *framework* e também uma ferramenta de *benchmark*, apresenta uma coleção mais variada de ferramentas e de algoritmos pré-estabelecidos de treinamento, com o intuito de servir de base comparativa.

3.4.1 Unity ML-Agents Toolkit

Figura 3.3: Imagem do jogo *Puppo*

A biblioteca chamada de *Unity ML-Agents* (JULIANI et al., 2018) nada mais é do que uma extensão do *software* chamado *Unity*, programa que possui um motor próprio de física tridimensional e que é voltado para o desenvolvimento de jogos eletrônicos. A extensão, proposta em 2018, fornece uma série de ferramentas para aqueles que querem

rodar simulações de problemas de inteligência artificial dentro do ambiente da *Unity*. Apesar de ter sido desenvolvida como uma plataforma de propósitos gerais, por causa da natureza do sistema sobre o qual ela foi incorporada, muitas das aplicações envolvem simulações de jogos, onde agentes aprendem a competir entre si.

Um dos princípios básicos que guiou a equipe responsável pelo desenvolvimento da *Unity ML-Agents Toolkit* foi possibilitar que desenvolvedores e estúdios de jogos eletrônicos incorporassem técnicas de inteligência artificial em suas obras utilizando técnicas atuais, como as redes neurais descritas na seção 2.4.2. Como forma de promover a plataforma e demonstrar suas capacidades, pesquisadores da Unity exibiram, em um evento em meados de 2018, o jogo *Puppo* (BERGES; CHEN, 2018), onde um cachorro busca um graveto atirado pelo jogador através de toques na tela (figura 3.3). O agente responsável pelos movimentos do cachorro, neste caso, foi treinado inteiramente utilizando-se de algoritmos de Aprendizagem por Reforço Profunda.

Dada a natureza do programa que serve de base para o simulador e o enfoque dado às ferramentas fornecidas por ele, fica claro que o interesse dos desenvolvedores é estimular o treinamento e uso de agentes nos jogos desenvolvidos na plataforma *Unity*. Problemas de robótica, por exemplo, são modelados e resolvidos de uma maneira muito mais simples e prática através de outras ferramentas, como a *Gazebo*. Fica claro, portanto, o foco dado ao *Unity ML-Agents Toolkit*: facilitar a vida daqueles que pretendem utilizar inteligência artificial no universo de jogos eletrônicos.

3.4.2 Gym

Em se tratando de aprendizado por reforço, ferramentas de *benchmark* servem para fornecer uma base padronizada de avaliação de algoritmos, o que significa que elas nada mais são do que um meio padronizado de comparar a performance de diferentes técnicas de aprendizado quando submetidas a um mesmo problema de aprendizado por reforço. Tais coleções permitem, por exemplo, que um algoritmo que está sendo desenvolvido seja comparado, em uma série de problemas, com diferentes outros algoritmos consagrados, a fim de que seja estudado o seu comportamento em diferentes situações.

Dado que problemas de aprendizado por reforço lidam com cenários onde um agente deve tomar uma série de ações em sequência, é natural que tenha surgido no ramo uma série de pesquisas e algoritmos que tentam ensinar agentes dotados de inteligência artificial a jogar diversos jogos eletrônicos. Duas coleções de cenários do tipo, que re-

produzem jogos famosos de plataformas antigas (ou seja, mais simples do que os jogos eletrônicos atuais e, portanto, mais fáceis de simular e de treinar agentes para jogá-los), chamadas de *VizDoom* e *ALE*, resolvem, até certo ponto, a necessidade de ter-se ferramentas de *benchmark* para problemas do tipo.

Desenvolvido pela OpenAI, o *Gym* (Brockman et al., 2016) é, em sua essência, um conjunto de simuladores de aprendizado por reforço que fornece ao pesquisador da área uma série de cenários (chamados de *environments*, ou ambientes) nos quais seus algoritmos de aprendizado podem ser desenvolvidos e testados. Já havia, antes da concepção do Gym, algumas outras ferramentas de *benchmarking* de AR, como o Arcade Learning Environment (BELLEMARE et al., 2013) ou o ViZDoom (KEMPKA et al., 2016), mas, além de reunir vários simuladores sob um único pacote, o Gym vai além e proporciona aos desenvolvedores que usam sua plataforma uma maneira de publicarem seus resultados, criando, desta forma, um ambiente colaborativo onde algoritmos e soluções são compartilhados. O objetivo dos idealizadores do Gym é criar, desta maneira, um ambiente onde desenvolvedores, estudantes e pesquisadores colaborem entre si, ao invés de competirem, encorajando que, junto com as soluções para cada problema, seja também fornecida uma breve explicação de como o problema foi abordado, quais técnicas foram usadas e como as informações fornecidas pelo ambiente foram interpretadas. Há ainda a possibilidade de ser submetido, juntamente do código e do pequeno texto explicativo, um vídeo ou imagem animada no formato *gif* do problema sendo resolvido, para ilustrar a evolução do treinamento e o resultado final atingido.

A coleção de cenários ofecerida pelo Gym é bastante abrangente, contendo problemas simples que podem ser descritos através de texto na tela do *console*, como o problema chamado *Frozen Lake*, retradado na figura 2.6, até problemas que utilizam motores de física (como o próprio MuJoCo, abordado na seção 3.3.2) para representar intrincados conjuntos de corpos e conexões. Por isso, não há apenas uma única maneira de representar, por exemplo, um estado: cenários mais simples podem conter menos informações a seu respeito ao mesmo tempo em que cenários que lidam com problemas de robótica possuem muito mais dados os quais podem ser lidos durante a fase de treinamento de um agente. Entretanto, todos os cenários do Gym foram construídos com base em algumas diretrizes simples, que tentam uniformizar, no que é possível, o processo de treinamento de um agente em um cenário disponibilizado pelo Gym.

Apesar, todavia, de apresentar uma ampla coleção de problemas que, além de possuírem uma estrutura padronizada, permitem que suas soluções sejam compartilhadas,

não há ferramentas que permitam a expansão desta coleção ou que auxiliem na criação de cenários novos. Tal possibilidade, como discutida no final da seção 3.4.2.1, é primordial para o desenvolvimento de algoritmos novos de aprendizado por reforço, uma vez que o desenvolvimento de soluções novas requer também a construção de novas ferramentas comuns de avaliação. Este trabalho apresenta uma ferramenta que auxilia na construção de novos cenários de aprendizado por reforço que, além de compatíveis com motores de física, também respeitam a estrutura ditada pelo *Gym*.

3.4.2.1 VizDoom e ALE

As ferramentas *VizDoom*, que simula o jogo Doom, e *ALE* (de *Arcade Learning Environment*), que simula dezenas de jogos *arcade* de plataformas como Atari, possuem o mesmo objetivo: fornecer uma série de cenários onde algoritmos de treinamento podem ser testados. As formas através das quais cada um dos *frameworks* atinge o seu objetivo, entretanto, é ligeiramente diferente: *VizDoom* simula a arquitetura do jogo *Doom*, tornando possível para o agente que informações como a distância entre o jogador e objetos ou monstros na cena seja acessada a qualquer momento. Isto torna o treinamento de agentes mais complexo, uma vez que o volume de dados que pode ser lido do ambiente simulado é maior. *ALE*, por sua vez, disponibiliza as informações do jogo ao agente apenas através de uma visualização da tela, que deve, por sua vez, ser alimentada a algum tipo de rede neural, como as DQNs, discutidas na seção 2.4.2.

Como ferramentas de *benchmark*, os dois *frameworks* cumprem bem o seu papel. Entretanto, com o avanço dos algoritmos estudados na área de AR, a necessidade de ferramentas de *benchmark* novas, também mais avançadas, surge juntamente com ele. A maioria dos cenários fornecidos pela *ALE*, por exemplo, já foi solucionada por algoritmos que apresentam performances bastante superiores à performance humana nos mesmos jogos, limitando o espaço disponível para algoritmos mais avançados demonstrarem sua performance (MACHADO et al., 2018). O desenvolvimento contínuo de ferramentas de *benchmkark*, portanto, deve acompanhar o desenvolvimento de novas soluções para problemas de aprendizado por reforço; há uma relação desproporcional, entretanto, entre o volume de estudos empregados no desenvolvimento de novos algoritmos e o montante de pesquisa destinada ao desenvolvimento de novas ferramentas de avaliação.

3.5 Arquitetura do Gym

Conforme descrito na seção 3.4.2, *Gym* é uma ferramenta que apresenta uma coleção de modelagens de problemas de aprendizado por reforço (chamadas de *cenários*) e que é mantida pela OpenAI, uma entidade sem fins lucrativos que promove estudos na área de inteligência artificial e cujo intuito é "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 fornecer 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 os simuladores do *Gym* um canal por onde se possa compartilhar ideias e comparar métodos.

É inegável que o compartilhamento de conhecimento entre pesquisadores de uma determinada área é essencial para alavancar o seu desenvolvimento. Entretanto, a escolha do canal ou da forma através dos quais este conhecimento é compartilhado pode muitas vezes limitar o seu alcance ou até mesmo impedir por completo que experimentos sejam replicados. O problema da reprodutibilidade é bastante conhecido: em uma pesquisa conduzida pela revista Nature (BAKER, 2016), 70% dos pesquisadores entrevistados admitiram que, ao menos uma vez, já tentaram, sem sucesso, reproduzir os experimentos de artigos escritos por terceiros, e mais da metade dos entrevistados disseram ter tido dificuldades em reproduzir os próprios experimentos. Do mesmo modo, também podem residir limitações no canal escolhido para divulgar um estudo novo: a grande maioria dos periódicos de artigos científicos é paga, o que pode vir a afastar o conhecimento daqueles que não têm dinheiro suficiente para pagar por ele.

Residem problemas, portanto, na forma com que o conhecimento derivado de estudos novos é passado adiante: nem sempre um artigo científico apresenta experimentos de fácil reprodução e nem sempre ele é publicado em um canal de fácil acesso ao grande público. Há uma tentativa, entretanto, por parte dos responsáveis pelo *Gym*, de atenuar este problema: ao estimular que soluções para os cenários oferecidos por ele sejam publicadas em um espaço aberto na rede, o acesso a algoritmos avançados de aprendizado por reforço é democratizado. A maneira através da qual o *Gym* propicia este compartilhamento de soluções é bem simples: há um guia, no formato de uma enciclopédia *on-line* e editável, que lista todos os cenários oferecidos pelo *Gym*, juntamente com as soluções de melhor performance. Há duas maneiras diferentes de se medir a performance de uma solução, de acordo com o cenário: em alguns cenários, o critério que qualifica as solu-

Figura 3.4: Classificação das melhores soluções para o problema *Mountain Car*.

User	Episodes before solve	Write-up	Video
BenSecret	8	writeup	
Udacity DRLND Team	13	writeup	gif
MisterTea, econti	24	writeup	
yingzwang	32	writeup	
sharvar	33	writeup	
SurenderHarsha	40	writeup	
n1try	85	writeup	
khev	96	writeup	video
ceteke	99	writeup	
manikanta	100	writeup	video
BS Haney	100	Write-up	YouTube
JamesUnicomb	145	writeup	video
Harshit Singh Lodha	265	writeup	gif
mbalunovic	306	writeup	
onimaru	355	writeup	video

ções é o numero de épocas de decisão necessárias para que o agente pudesse solucionar o problema — e aqui a definição de "solução"é particular de cada cenário: no problema chamado de *Cart Pole*, por exemplo, o problema é considerado solucionado quando a recompensa acumulada de um episódio ultrapassa uma determinada marca — e em outros problemas, estes mais complexos e, portanto, mais difíceis de definir quando o problema foi de fato solucionado, usam como critério de avaliação de soluções apenas a recompensa média recebida em cem episódios. Os números são facilmente verificáveis, uma vez que, juntamente com um pequeno texto que explica o funcionamento do algoritmo, deve ser também submetido o código utilizado na solução, com ambos ficando disponíveis para *download*. Em alguns casos, também é encorajada a submissão de um vídeo ou imagem no formato *GIF* do comportamento do agente, quando da solução do problema. Um exemplo de listagem das melhores soluções para um problema da coleção do *Gym* é dado na figura 3.4.

3.6 Implementação do Gym

Além de atacar o problema resultante da dificuldade em compartilhar-se resultados de um experimento, o *Gym* propõe uma espécie de padronização de problemas de

aprendizado por reforço, que fica evidente quando é analisada a maneira através da qual os diferentes problemas de aprendizado por reforço são transformados em cenários do *Gym*. Na seção a seguir, será estudada a implementação do *Gym*, para que fique claro de que maneira esta padronização é feita.

3.6.1 Representação do ciclo de aprendizado

A implementação do *Gym* é bastante próxima do ciclo de aprendizado de um agente de AR definido na imagem 2.5, baseando-se apenas nos seguintes conceitos de aprendizado por reforço:

- o agente está inserido em um ambiente;
- a cada passo, o agente toma uma *decisão* e recebe uma *recompensa* e realiza uma *observação* do ambiente;
- o ambiente é representado através de Processo de Decisão de Markov Parcialmente Observável (POMDP) (SUTTON; BARTO, 2018).

Baseando-se nos conceitos acima, o grande destaque do *Gym* talvez seja no sistema de *episódios* de um treinamento de uma gente de AR. Quando um treinamento de um agente separa-se em episódios, um cenário é iniciado em um estado randômico e o agente toma uma série de decisões até que uma ação modifique o ambiente de tal forma que ele transicione para um estado terminal. Quando isso ocorre, é calculada a recompensa total, acumulada durante toda a sequência de ações tomadas pelo agente desde o momento em que o cenário foi inicializado até o momento em que ele chegou em um estado terminal, e o cenário é reinicializado. O código a seguir exemplifica a inicialização de um agente, que toma uma série de ações até que um estado final seja atingido:

```
1 # definição do cenário
2 env = gym.make('CartPole-v0')
3
4 # cenário é inicializado, observação retornada
5 obs = env.reset()
6 done = False
7
8 while not done:
9   # observação, recompensa para a ação tomada
```

```
10    # e variável booleana indicando se cenário
11    # chegou a um estado terminal são retornados
12    obs, reward, done, _ = env.step(agent.act(observation))
```

É necessário observar aqui que toda a modelagem de um cenário disponibilizado pelo *Gym* gira em torno do ambiente e das transições entre seus estados. O agente, apesar de ser um conceito bastante importante em problemas de aprendizado por reforço, não possui uma abstração explícita no *Gym* (no exemplo acima, um agente, desenvolvido à parte, age de acordo com a observação), permitindo, desta maneira, que ele seja modelado da maneira que for mais conveniente para quem desenvolve soluções para um determinado cenário. Toda informação relevante para a solução do problema é retornada durante uma tomada de decisão (exemplificada na linha 12 do código acima) e segue o formato $(observation, reward, done, info)$, onde:

- $observation$ é um vetor de números que representa o estado do ambiente naquele momento;
- $reward$ é a recompensa atribuída à ação tomada;
- $done$ é uma variável binária que diz se o estado atual do ambiente é um estado terminal, utilizada para reiniciar as simulações quando assim for conveniente;
- $info$ é um dicionário com informações adicionais sobre o episódio, utilizado apenas quando necessário.

Este formato, através do qual as informações do cenário são representadas, juntamente com a ausência de uma abstração da entidade tomadora de decisões, permite vários estilos diferentes de aprendizado. As informações retornadas a cada n episódios, por exemplo, podem ser agrupadas em um lote de dados que só mais tarde, após finalizados os episódios, é repassado a uma entidade de aprendizado por reforço que processa as informações e atualiza o seu agente de acordo. Um outro estilo de aprendizado, por exemplo, pode consistir em utilizar as informações de cada episódio para o treinamento do agente no momento em que elas são fornecidas, corrigindo o seu comportamento de maneira incremental. Além disso, a ausência de um modelo de agente permite que sejam desenvolvidos cenários onde dois ou mais agentes atuam.

A decisão de subtrair do sistema a abstração de um agente, portanto, dá mais liberdade para pesquisadores desenvolverem agentes com a interface que for mais conveniente. Com o agente transformado em uma espécie de "caixa-preta" no ecossistema do *Gym* é bastante simples que diferentes agentes sejam utilizados alternadamente a fim de que suas

performances sejam comparadas utilizando-se de um mesmo cenário de base. É possível também que um agente seja pré-treinado em um problema antes de ser inserido em outro cenário, permitindo que o conhecimento adquirido no primeiro cenário seja utilizado no segundo, técnica conhecida como *transfer learning* (TORREY; SHAVLIK, 2010).

3.7 Representação dos resultados

Figura 3.5: Classificação das melhores soluções para o problema *Pendulum-v0*.

User	Best 100-episode performance	Write-up	Video
msinto93	-123.11 ± 6.86	D4PG	
msinto93	-123.79 ± 6.90	DDPG	
heerad	-134.48 ± 9.07	writeup	
BS Haney	-135	Write-up	YouTube
ThyrixYang	-136.16 ± 11.97	writeup	
lirnli	-152.24 ± 10.87	writeup	

Além de versar sobre a forma como os cenários são representados no *Gym*, é importante também falar sobre como os resultados de um treinamento são gravados. A performance de um treinamento pode ser medida de acordo com dois critérios: o primeiro é a quantidade de episódios necessários para que o problema seja dado como resolvido, critério este que varia de acordo com o cenário. Nem todos os cenários, entretanto, possuem um mecanismo que informa explicitamente se o problema foi resolvido; há, portanto, um meio alternativo de se medir a performance de um treinamento: usa-se, neste caso, os melhores valores de recompensas totais recebidas para uma série de cem episódios consecutivos, como ilustrado na figura 3.5. Para que seja facilitada a comparação de resultados, o *Gym* possui uma estrutura interna chamada de *Monitor*, que mantém um controle interno de todas as vezes que o uma simulação é atualizada ou reiniciada. Esta estrutura, que também pode gravar videos periodicamente, torna possível a construção de curvas de aprendizado, a partir dos dados oriundos das recompensas recebidas. O código a seguir ilustra uma situação onde o *Gym* é programado para gravar vídeos do treinamento em uma pasta chamada *recording*.

```
1 env = gym.make("CartPole-v0")
2 env = gym.wrappers.Monitor(env, "recording")
```

3.7.1 Arquitetura de um cenário

Depois de explicitados os meios através dos quais o *Gym* representa as informações do ambiente, como o agente lê essas informações e como os resultados do seu aprendizado são arquivados para análises futuras, resta saber como um cenário é modelado no sistema do *Gym*. Em (HESSE, 2019), é dada uma descrição detalhada de todos os arquivos que devem compor um cenário e qual deve ser o conteúdo deles. Deve ser observado que, para o arquivo que contém a classe do ambiente, devem ser implementados obrigatoriamente os seguintes métodos:

- `observation()`: método que deve retornar a observação do ambiente, que nada mais é do que o conjunto de informações que o agente usará para tomar sua decisão e que corresponde à "observação do estado atual", na figura 2.5;
- `action()`: método que processa a ação tomada pelo agente e que corresponde à "tomada de decisão"da figura 2.5. Deve também processar as transições entre estados;
- `reward()`: método que computa a ação a ser recebida pelo agente, correspondente ao "recebimento de recompensa"da figura 2.5;
- `done()`: método que diz se o estado atual da simulação é um estado terminal. Corresponde ao "Estado atual é terminal?"da figura 2.5.

No próximo capítulo, serão expostos as dificuldades encontradas na hora de escolher a ferramenta correta para a construção de simuladores de aprendizado por reforço. Também será exposto o Barbell, que é uma tentativa de contornar algumas dessas dificuldades e também de oferecer vantagens para aqueles que pretendem construir simulações.

4 BARBELL

Conforme desenvolvido nos capítulos anteriores, técnicas de aprendizado por reforço são aplicáveis a uma vasta gama de problemas, dado que existem inúmeras situações onde um problema deve ser resolvido através de ações tomadas sequencialmente. Seja definir quais jogadas devem ser tomadas em um tabuleiro de xadrez para que a partida seja vencida ou decidir como regular a força propulsora das hélices de um veículo aéreo autocontrolado a fim de torná-lo estável diante de mudanças na direção do vento, quaisquer problemas que requeiram a tomada sequencial de ações e nos quais a qualidade dessas ações possam ser captadas pela noção de recompensa permitem o uso de algoritmos de AR. Também foi extensamente discorrindo e exemplificado sobre a importância do uso de ferramentas de simulação no treinamento de agentes de aprendizado por reforço, mostrando que, apesar das diversas limitações que um simulador pode possuir, tais ferramentas fornecem uma maneira poderosa de modelar um problema, deixando as diferentes situações às quais o agente é submetido sob o controle daqueles que o treinam.

Além do papel imprescindível que ferramentas de modelagem de problemas de aprendizado por reforço exercem durante a fase de treinamento de agentes para resolvê-los, também foi discorrido acerca dos diferentes graus de especificidade que tais *softwares* podem apresentar. Para problemas de uma subárea específica de aprendizado por reforço como a robótica, por exemplo, pode ser mais vantajoso para quem deseja treinar um agente adotar ferramentas menos genéricas de construção de ambientes em detrimento de ferramentas de propósito mais geral, dado que *frameworks* mais específicos fornecem atalhos para a modelagem de problemas que não fogem ao seu escopo. Sendo assim, há de ser analisado o *trade-off* que surge do conflito entre a adoção de um mecanismo de modelagem de simulações de propósito mais geral, que permite a construção de problemas das mais variadas naturezas mas que não fornece atalhos que facilitem a criação de simulações de uma área em específico, e a adoção de ferramentas mais especializadas, como o *MuJoCo*, voltado para a robótica, ou a *Unity ML-Agents Toolkit*, voltada para problemas envolvendo jogos onde agentes competem entre si, que fornecem caminhos mais curtos e ferramentas que facilitam a criação de cenários dentro dos seus próprios contextos.

Além da necessidade de haver programas de simulação com diferentes graus de especificidade, também versamos neste trabalho sobre a necessidade de ter-se meios comuns de avaliação de algoritmos: as chamadas ferramentas de *benchmark*. Tais mecanismos de avaliação de algoritmos fornecem uma base comum de comparação para aqueles

que propõem novos métodos de aprendizado por reforço, permitindo que estes tenham sua performance facilmente posta à prova quando aplicados em cenários ou problemas amplamente conhecidos e estudados. Assim como apresentado na discussão acerca dos diferentes graus de especificidade que um instrumento de simulação de problemas de aprendizado por reforço pode ter, há aqui também um certo grau de limitação: como discutido em (Brockman et al., 2016), o desenvolvimento de novos algoritmos de AR depende diretamente do constante criação de coleções de problemas de *benchmark*, dado que tais coleções tendem a tornarem-se praticamente inúteis à medida em que o campo do aprendizado por reforço avança e os problemas fornecidos por elas são resolvidos.

A ferramenta de *benchmark* analisada por este trabalho, talvez a mais famosa delas — e certamente a que possui a coleção de problemas mais ampla — seja o *framework* conhecido como *Gym*. Muito mais do que uma ferramenta de *benchmark* com uma vasta coleção de diferentes cenários que compreendem desde problemas simples, com representação em modo texto, até problemas que envolvem robôs que devem aprender a andar em universos virtuais regidos por um motor de física, *Gym* consegue ir além e fornecer meios de compartilhamento de soluções para os problemas oferecidos, estimulando a colaboração entre pesquisadores do mundo todo.

Apesar da vasta coleção oferecida pelo *Gym* e pela facilidade com que diferentes soluções para um mesmo problema podem ser comparadas e compartilhadas, há poucas ferramentas disponíveis para a a criação e o compartilhamento de cenários novos. A criação de cenários novos para ferramentas de *benchmark* é tão importante quanto o desenvolvimento de novos métodos de aprendizado, apesar de não receber tanta atenção por parte dos especialistas como a construção de novos algoritmos. O campo carece, portanto, de uma ferramenta que permita o desenvolvimento rápido de novos cenários para a expansão de coleções de *benchmark*.

Este trabalho é uma tentativa de estabelecer um meio para a criação de novos cenários de aprendizado por reforço. Além de fornecer um meio simples para a modelagem de problemas de qualquer natureza, tentando, quando possível, agregar elementos presentes em ferramentas de simulação de alto grau de especificidade e que funcionam como atalhos na criação de cenários, o trabalho aqui desenvolvido produz cenários compatíveis com a API do *Gym*, beneficiando-se, desta maneira, das ferramentas disponibilizadas pela plataforma para o compartilhamento, a avaliação e a comparação de diferentes soluções para um mesmo problema.

O *Barbell*, *software* proposto por este trabalho, é, portanto, uma espécie de ex-

tensão ao *Gym*, que permite que sua coleção de problemas seja expandida através de ferramentas que visam facilitar a construção de novos cenários de aprendizado por reforço. Ao mesmo tempo que tenta empregar as vantangens oferecidas por simuladores altamente específicos, como a descrição textual de elementos a serem simulados no ambiente, o *Barbell* tenta manter a natureza genérica do *Gym*, aplicando, desta maneira, as vantagens de simuladores de alto grau de especificidade na modelagem de problemas das mais diversas naturezas.

O *Barbell*, em sua essência, nada mais é do que um gerador de código. A sua maior vantagem, entretanto, é a geração de código compatível com o *Gym*, permitindo não só que os resultados produzidos por ele sejam compatíveis a API proposta pelo *Gym*, mas também que seus usuários não precisem desprender muito tempo no desenvolvimento de seus cenários de simulação.

A seguir, as funcionalidades do *Barbell* serão apresentadas a fim de que fique claro para o leitor quais funcionalidades são estas e de que maneira elas se beneficiam daquilo que o *Gym* já fornece. O *Barbell* conta com duas ferramentas de geração de código, uma para gerar o código referente à interface do programa, e outra que permite que seja gerado o código da simulação do problema.

4.1 Gerador de código de interface

O gerador de código para interface é, basicamente, uma ferramenta que gera a estrutura formada pelos métodos que implementam o ciclo de aprendizado descrito na figura 2.5, comum a todos os ambientes implementados pelo Gym e descrita na seção 3.6. Além de gerar um arquivo descritivo com informações a respeito do ambiente coletadas através de uma ferramenta interativa de linha de comando, o gerador de código de interface do Barbell é capaz de gerar variações de um mesmo ambiente, o que é comum no Gym.

A seguir, um exemplo de ambiente chamado `Exemplo1` gerado pelo Barbell:

```
1 import gym
2 from barbell_environment import BarbellWorld, BarbellViewer, \
3                                 BarbellContact, BarbellStatistics
4 from barbell_utils import parse_file
5
6 # CONSTANTS
7 FPS = 50  # desired FPS rate
8
```

```
9
10  class Barbelltest1ContactDetector(BarbellContact):
11      def __init__(self, env):
12          BarbellContact.__init__(self)
13          self.env = env
14
15      def BeginContact(self, contact):
16          pass
17
18      def EndContact(self, contact):
19          pass
20
21
22  class Barbelltest1(gym.Env):
23      metadata = {
24          'render.modes': ['human', 'rgb_array'],
25          'video.frames_per_second': FPS
26      }
27
28      def __init__(self):
29          self.viewer = None
30          self.viewport_width = None
31          self.viewport_height = None
32          self.gravity = None
33          self.partslist = {}
34          self.jointslist = []
35          self.name = 'barbelltest1-v0'
36          self.world = None
37          self.current_step = 0
38          self.current_episode = 0
39          self.statistics = False
40          self.statisticsRecorder = None
41
42          self.initialize_world({})
43
44      def initialize_world(self, objects):
45          if self.world is not None:
46              for body in self.world.objects:
47                  self.world.DestroyBody(self.world.objects[body])
48          self.world = BarbellWorld(gravity=self.gravity)
49          self.world.initialized_contact_detector = Barbelltest1ContactDetector(self)
```

```
50          self.world.contactListener = self.world.initialized_contact_detector
51
52      def get_object(self, object_name):
53          return self.world.objects[object_name]
54
55      def get_joint(self, object_a, object_b):
56          return self.world.joints["%s_%s" % object_a, object_b]
57
58      def set_definition_file(self, filename):
59          parse_file(filename, self)
60
61      # FILL ME IN!
62      def reset(self):
63          self.current_step = 0
64          self.total_reward = 0
65          self.current_episode += 1
66
67      # FILL ME IN!
68      def observation(self):
69          pass
70
71      # FILL ME IN!
72      def reward(self):
73          pass
74
75      # FILL ME IN!
76      def done(self):
77          pass
78
79      # FILL ME IN!
80      def step(self, action):
81          reward = self.reward()
82          self.total_reward += reward
83          observation = self.observation()
84          done = self.done()
85          self.world.Step(1.0 / FPS, 6 * 30, 2 * 30)
86          if done and self.statistics:
87              if self.statisticsRecorder is None:
88                  self.statisticsRecorder = BarbellStatistics(self.env_name)
89              self.statisticsRecorder.save(self.current_epoch, self.total_reward)
90          return observation, reward, done, {}
```

```
91
92      def render(self, mode='human', close=False):
93          if self.viewer is None:
94                      self.viewer = BarbellViewer(
95                        viewport_width=self.viewport_width,
96                        viewport_height=self.viewport_height)
97
98          self.viewer.draw_objects(self.world.objects)
99          return self.viewer.render(return_rgb_array=(mode == 'rgb_array'))
```

Código-fonte 4.1: Exemplo de código gerado para o Barbell.

A ferramenta de geração de código para interface, além de criar todos os arquivos que um ambiente do Gym deve conter, gera trechos de código muito importantes para a simulação de um problema de aprendizado por reforço. No código 4.1, que contém apenas as linhas geradas automaticamente pelo Barbell, isto é bastante visível: a classe `BarbellTest1`, gerada para um ambiente de mesmo nome, contém, em seu método inicializador (linha 28 a 43), a declaração de informações da simulação que, de outro modo, teriam que ser inicializadas manualmente, como a episódio de treinamento atual (linha 38) e o instante de tempo do episódio atual (linha 37). O método `initialize_world()` (linhas 44 a 50), chamado ao final do inicializador, fornece as ferramentas de criação do ambiente de simulação pelo motor de física (neste caso, o Box2D). Neste trecho de código também é criado uma instância do detector de contato, declarado na linha 10, que possui métodos de *callback* que são chamados sempre que dois objetos colidem.

Nas linhas seguintes, há os métodos que devem ser completados. Cada um deles cumpre uma função necessária para a implementação do fluxo de aprendizado descrito na figura 2.5. São eles:

- `reset()`: método que é chamado que seja preparado o início de um episódio de simulação. A criação dos objetos presentes no cenário inicial de um episódio de simulação, por exemplo, deve ser feita aqui.
- `observation()`: método que, a cada passo da simulação, deve retornar o vetor de informações que compõem a observação do estado atual da simualação. Este vetor pode conter informações como a localização atual de um objeto ou sua velocidade, por exemplo.
- `reward()`: método que, a cada passo da simulação, deve retornar a recompensa referente àquele passo. A soma das recompensas retornadas durante um episódio da simulação está na variável `total_reward`, criada e zerada automaticamente

sempre que a simulação é reiniciada.

- `done()`: método que, a cada passo da simulação, deve retornar um valor booleano que diz se a simulação encontra-se em um estado terminal. Em caso positivo, a simulação deve ser reiniciada através de uma chamada ao método `reset()`.
- `step()`: método responsável por executar o código necessário para processar as transições entre os estados da simulação e, portanto, seu andamento. Recebe como parâmetro uma ação e cabe ao desenvolvedor definir como a ação será interpretada; a transição entre estados deve, entretanto, ocorrer obrigatoriamente. Deve devolver o vetor de observação do novo estado, a recompensa relativa à ação informada e uma variável booleana que diz se o novo estado é um estado terminal, variável esta que é, por sua vez, utilizada para definir se o métódo `reset()` será chamado. Há, além das variáveis `observation`. `reward` e `done`, um dicionário que deve ser retornado. Isto faz parte do padrão dos ambientes do Gym e esta estrutura pode ser utilizada para retornar informações adicionais da simulação que não serão utilizadas pelo agente.

Há ainda o método `render()`, responsável pela renderização da simulação e o método `set_definition_file()`, que lê um arquivo YAML com definições de objetos e juntas e os guarda nas variáveis `partslist` e `jointslist`, respectivamente. Para que os objetos declarados no arquivo sejam efetivamente criados, os métodos `create_object()` e `create_joint()` devem ser chamados, e os objetos a serem criados devem ser passados como parâmetro. A criação de um arquivo com definições de objetos e juntas é puramente opcional, visto que o carregamento do arquivo resulta no uso de funções da API do Barbell que podem ser chamadas diretamente. O uso de um arquivo externo para definir os objetos que serão criados na simulação oferece, entretanto, três vantagens: a primeira é a legibilidade de código: o código torna-se mais claro se separarmos a lógica da simulação, que trata de todos os passos do ciclo de aprendizado, da lógica dos objetos que fazem parte da simulação. A segunda vantagem trazida pelo arquivo é a capacidade de reuso de código: um objeto definido apenas uma vez no arquivo YAML pode ser modificado e reutilizado quantas vezes for necessário durante todo o ciclo da simulação. Em simulações que envolvem diversas variações de um mesmo elemento, como normalmente é o caso em simualações de *swarm intelligence*, por exemplo, um grupo de elementos iguais requer apenas uma única definição. A terceira vantagem é a clareza de código: a estrutura utilizada para declarar objetos e juntas no arquivo YAML é uma linguagem de alto nível bastante descritiva, o que é bastante diferente das funções

de motores de física que implementam corpos e conexões.

O gerador de código de interface é bastante útil para gerar simulações compatíveis com a biblioteca do Gym, montando toda a estrutura comum de arquivos e trazendo algumas estruturas de controle que são praticamente onipresentes em simulações do tipo, como os métodos expostos nesta seção. A possibilidade de declarar objetos e juntas em um arquivo à parte, porém, separando a lógica da simulação e a definição dos objetos que a compõem, permitindo o reuso de definições, é o que fornece os maiores atalhos para quem pensa em construir cenários para o Gym. As ferramentas de criação de objetos e juntas serão expostas e explicadas na seção a seguir.

4.2 Gerador de código de simulação

O gerador de código de simulação é, essencialmente, um módulo do Barbell capaz de ler um arquivo YAML que contém, em uma linguagem descritiva e de alto nível, a definição dos objetos que irão compor a simulação e as juntas que os conectarão, além de algumas outras definições básicas como o tamanho da tela e o fator de escala entre pixels (unidade de medida da janela onde a simulação é renderizada) e metros (unidade utilizada pelo motor de física). Há três seções básicas no arquivo: `DOMAIN`, `PARTS` e `JOINTS`. Na seção `DOMAIN`, podem ser declarados valores sob as seguintes chaves:

- `viewport_width`: largura da tela onde será renderizada a simulação;
- `viewport_height`: altura da tela onde será renderizada a simulação;
- `ppm`: pixels por metro, fator de escala entre pixels e metros;
- `statistics`: variável booleana que diz que os resultados devem ser salvos ou não (ver seção 4.3).

No exemplo 4.2, que define uma simulação de um carro com duas rodas, todas estas opções são utilizadas. A parte mais importante do código, entretanto, começa na linha 8, com a definição das partes do carro (um corpo retangular e duas rodas) e as juntas que os conectam. No Barbell, há duas opções quanto ao tipo do corpo: estático, que indica que um corpo permanecerá durante toda a simulação com as suas propriedades físicas iniciais (como a sua posição, por exemplo), e dinâmico, que indica que um corpo sofrerá as consequências de forças físicas aplicadas a ele.

Quanto ao formato, há três tipo de formato que podem ser declarados: `circle`, `polygon` e `box`, que indicam que um corpo é esférico (no caso da simulação bidimen-

sional, que o corpo é representado por um círculo), formado por uma série de vértices, ou que é quadrado, respectivamente — este último sendo apenas um atalho para definir corpos quadrilaterais, que podem ser declarados através de uma sequência de vértices. Os corpos possuem propriedades como densidade e fricção, que são utilizadas pelo motor de física para calcular as forças resultantes das interações entre os objetos. Além disso, cada objeto possui cores, declaradas através de seu nome em inglês ou por uma lista de três números $v \in [0, 1]$ que indicam uma cor no sistema RGB.

Os corpos declarados não são criados dentro do universo simulado automaticamente. Ao invés disso, ficam salvos em uma estrutura salva na variável `partslist`, fazendo com que os corpos possam ser modificados em tempo de execução antes de serem criados de fato. Isto permite, por exemplo, que em uma simulação onde há varias variações de um mesmo corpo, este seja declarado apenas uma vez, e os detalhes de cada uma das cópias que as diferenciam das demais sejam atribuídos a elas em tempo de execução. O uso de um arquivo YAML para declarar objetos é, portanto, opcional. Seu uso, entretanto, mantém separadas as lógicas referentes aos objetos que irão compor a simulação com a lógica que diz respeito aos passos da simulação, o que torna o código mais claro uma vez que o código que define as propriedades de cada corpo e o código que trata dos passos do ciclo de aprendizado não estão misturados em um mesmo arquivo. Além disso, o motor de física escolhido faz uma separação entre a estrutura que define um corpo (chamada de `body`) e que possui propriedades físicas estáticas ou dinâmicas e a estrutura que define um objeto, chamada de `fixture`, que está ligada à forma que este corpo assume, seja ele um círculo ou um conjunto de vértices. No Barbell, entretanto, esta separação não existe, simplificando a declaração de corpos e objetos, unindo-as.

O código-fonte 4.3 mostra o que seria o equivalente às definições do código 4.2 se declaradas diretamente no ambiente do Box2D. É bastante perceptível o grau de eloquência das declarações feitas em YAML ante às mesmas definições feitas diretamente no Box2D. Além disso, o código 4.3 não trata da parte visual da simulação, que é gerada automaticamente pelo Barbell.

```
1 DOMAIN:
2   viewport_width: 400
3   viewport_height: 200
4   ppm: 50
5   statistics: True
6
7
```

```yaml
PARTS:
  car:
      body_type: dynamic
      body_shape: box
      initial_position: [100, 60]
      color1: blue
      color2: black
      box_size: [80, 30]
  wheel1:
      body_type: dynamic
      body_shape: circle
      initial_position: [500, 600]
      radius: 20
      color1: black
      color2: blue
      z_index: 1
  wheel2:
      body_type: dynamic
      body_shape: circle
      initial_position: [500, 500]
      radius: 20
      color1: black
      color2: blue
      z_index: 1
  floor:
      body_type: static
      body_shape: box
      initial_position: [600, 5]
      color1: black
      box_size: [600, 5]


JOINTS:
    - connects: [car, wheel1]
      type: revolute
      local_anchor_a: [-80, -35]
    - connects: [car, wheel2]
      type: revolute
      local_anchor_a: [80, -35]
```

Código-fonte 4.2: Definição YAML do cartpole

```python
from Box2D import b2World, b2Vec2
```

```
2
3  PPM = 50
4  world = b2World(gravity=(0, -10))
5
6  car = world.CreateDynamicBody(position=b2Vec2(100, 60) / PPM, angle=0)
7  car.CreatePolygonFixture(box=b2Vec2(80, 30) / PPM,
8                           density=1,
9                           friction=0.25)
10 car.color1 = [0, 0, 1]
11
12 wheel1 = world.CreateDynamicBody(position=b2Vec2(100, 20) / PPM, angle=0)
13 wheel1.CreateCircleFixture(radius=20 / PPM, density=1, friction=0.25)
14 wheel1.color1 = (0, 0, 1)
15 wheel1.color2 = (0, 0, 0)
16
17 wheel2 = world.CreateDynamicBody(position=b2Vec2(260, 20) / PPM, angle=0)
18 wheel2.CreateCircleFixture(radius=20 / PPM, density=1, friction=0.25)
19 wheel2.color1 = (0, 0, 1)
20 wheel2.color2 = (0, 0, 0)
21
22 floor = world.createStaticBody(position=b2Vec2(600, 5) / PPM, angle=0)
23 floor.createPolygonFixture(box=b2Vec2(600, 5) / PPM, density=1, friction=0.25)
24
25 world.createRevoluteJoint(car, wheel1, localAnchorA=b2Vec2(-80, -35) / PPM,
26                            localAnchorB=(0, 0))
27 world.createRevoluteJoint(car, wheel2, localAnchorA=b2Vec2(80, -35) / PPM,
28                            localAnchorB=(0, 0))
```

Código-fonte 4.3: Definição do Cartpole em Box2D

No Barbell, a definição de juntas é bastante parecida com a definição de objetos: lista-se as juntas que farão parte da simulação, indicando quais corpos elas conectarão, e as mesmas serão armazenadas na variável `jointslist`. Há nove tipos diferentes de juntas que podem ser criadas no ambiente do Box2D. Para criá-las no Barbell, basta indicar qual o tipo de junta está sendo criada usando a chave `type` que todas as demais chaves (com a exceção de `connects`, que indica quais objetos serão conectados e que é interna ao Barbell) serão passadas como parâmetro para a função correspondente do motor de física. Uma trabalho sobre as características de cada tipo de junta pode ser encontrado em (SHEN, 2015) e a documentação completa pode ser encontrada em (CATTO, 2007).

4.3 Gerador de resultados

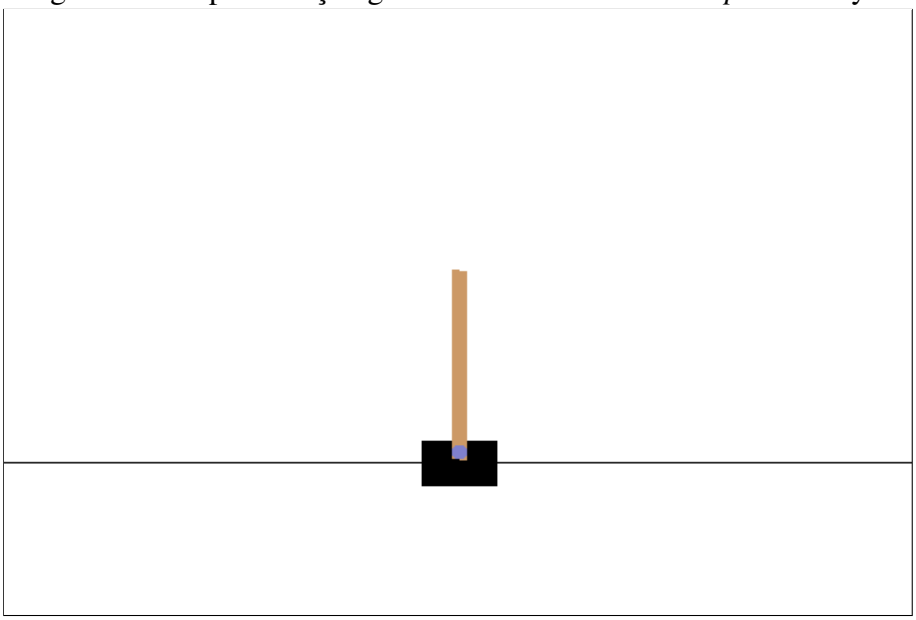
O gerador de resultados, como o nome indica, não se trata de um gerador de código, mas de um módulo do Barbell responsável por gravar os resultados das simulações. No código-fonte 4.1, nas linhas 86 a 89, há o trecho que faz uso dele: caso a variável `statistics` carregue em si um valor positivo e o passo atual seja terminal, é gravado em um arquivo CSV uma linha com o número que indica qual episódio de treinamento acabou de terminar e qual foi a recompensa total recebida em seus passos de execução. No experimento da seção 5.1, há um exemplo de uso dos dados gerados durante o treinamento através deste módulo. O acionamento do módulo se dá através de um valor booleano sob a chave `statistics`, declarado no arquivo YAML, e seu valor padrão é falso, ou seja, o módulo não grava os resultados por padrão.

5 EXPERIMENTOS

Para ilustrar as capacidades do *Barbell*, foram realizados três experimentos. Os experimentos simulam problemas clássicos da literatura e um jogo de grande sucesso, que também foi alvo de estudos na área de aprendizado por reforço. Dois dos três problemas apresentados também são disponibilizados nativamente pela coleção do Gym. Para fins de comparação, o código que os implementa encontra-se nos anexos deste trabalho.

5.1 Cartpole

Figura 5.1: Representação gráfica do ambiente do *Cartpole* no Gym



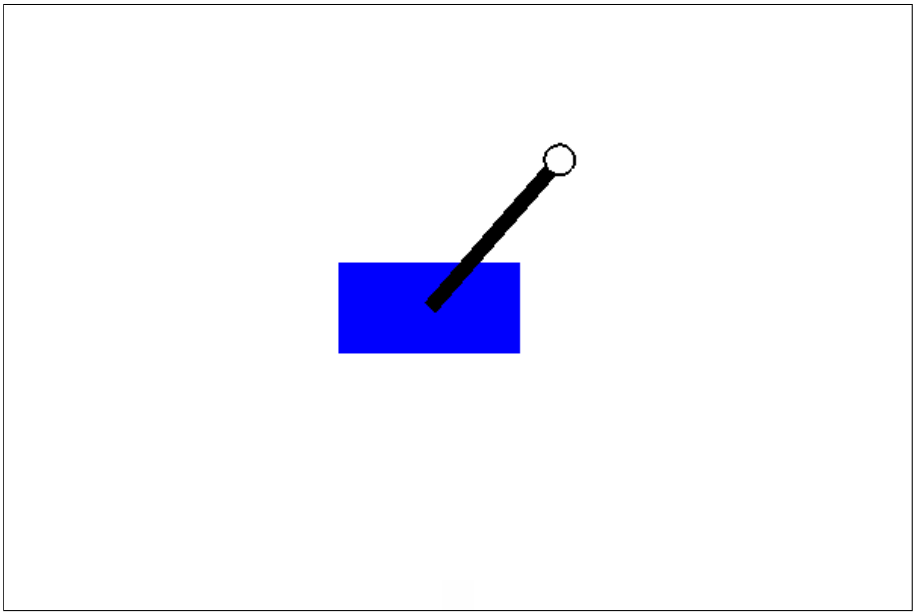
O problema conhecido como *Cartpole* (BARTO; SUTTON; ANDERSON, 1983) é simples: nele, há uma espécie de carrinho com uma haste acoplada em seu corpo, e o carrinho deve se mover pelo trilho a fim de manter a haste equilibrada, apontando para cima. O agente deve, portanto, aprender a controlar o carrinho, movendo-o para um lado ou para o outro, para impedir que a haste caia (figura 5.1). A formulação do problema é bastante simples: um episódio de treinamento dura até que o ângulo da haste seja maior do que um ângulo θ ou até o carro esteja próximo às bordas da tela. A recompensa é de $+1$ para cada instante de tempo que a haste permanece em pé e a ação é representada por um número que pode ser 0 (carrinho move-se para a esquerda) ou 1 (carrinho move-se para a direita). Por sua vez, o estado da simulação é representado por um vetor de quatro

elementos formado por:

- Posição x do carrinho;
- velocidade atual do carrinho;
- ângulo da haste;
- velocidade angular da haste.

O código gerado pelo Barbell que implementa o ambiente *Cartpole* é dividido em um arquivo em formato YAML de definição dos objetos e um *script* que implementa os passos da figura 2.5. O resultado da simulação pode ser visto na figura 5.2 e os arquivos que a definem são listados a seguir.

Figura 5.2: Representação gráfica do ambiente do *cartpole* no Barbell



```
import random
import numpy as np
import gym
from gym import spaces
from barbell_environment import BarbellWorld, BarbellViewer, BarbellStatistics
from barbell_utils import parse_file

DEG_TO_RAD = 0.0174533
RAD_TO_DEG = 57.2958
FPS = 50  # desired FPS rate


class Pendulum1(gym.Env):
```

```
14      metadata = {
15          'render.modes': ['human', 'rgb_array'],
16          'video.frames_per_second': FPS
17      }
18
19      def __init__(self):
20          self.current_epoch = 0
21          self.viewer = None
22          self.viewport_width = None
23          self.viewport_height = None
24          self.gravity = None
25          self.partslist = {}
26          self.jointslist = []
27          self.statistics = False
28          self.statisticsRecorder = None
29          self.total_reward = 0
30          self.current_epoch = 0
31          self.env_name = 'cartpoleBarbell-v0'
32          self.world = BarbellWorld(gravity=self.gravity)
33
34          self.action_space = spaces.Discrete(2)
35          high = np.array([
36              self.viewport_width,
37              np.finfo(np.float32).max,
38              40 + 5,
39              np.finfo(np.float32).max])
40
41          low = np.array([
42              0,
43              np.finfo(np.float32).max,
44              -40 - 5,
45              np.finfo(np.float32).max])
46
47          self.observation_space = spaces.Box(low, high, dtype=np.float32)
48
49      def get_object(self, object_name):
50          return self.world.objects[object_name]
51
52      def get_joint(self, object_a, object_b):
53          return self.world.joints["%s_%s" % object_a, object_b]
54
```

```
55      def set_definition_file(self, filename):
56          parse_file(filename, self)
57
58      def reset(self):
59          self.current_epoch += 1
60          self.current_step = 0
61          self.total_reward = 0
62          for body in self.world.objects:
63              self.world.DestroyBody(self.world.objects[body])
64          self.world = BarbellWorld(gravity=self.gravity)
65
66          self.world.create_objects(self.partslist)
67          self.world.create_joints(self.jointslist)
68          self.world.apply_force('local', 'pole',
69                 (random.choice((1, 1)) * random.choice((1, 2, 3, 4)), 0))
70
71      def observation(self):
72          obs = [
73              self.get_object('cart').position[0] * self.ppm,
74              self.get_object('cart').linearVelocity[0],
75              self.get_object('pole').angle * RAD_TO_DEG,
76              self.get_object('poletop').linearVelocity[0]
77          ]
78          return obs
79
80      def done(self):
81          angle = abs(self.get_object('pole').angle)
82          if angle * RAD_TO_DEG >= 40 or self.current_step == 300:
83              return True
84          else:
85              return False
86
87      def step(self, action):
88          if action == 0:
89              self.world.apply_force('local', 'cart', (-3, 0))
90          elif action == 1:
91              self.world.apply_force('local', 'cart', (3, 0))
92
93
94          if done:
95              reward = 0
```

```
96          else:
97              reward = 1
98
99          observation = self.observation()
100         done = self.done()
101         self.total_reward += reward
102         self.current_step += 1
103         self.world.Step(1.0 / FPS, 6 * 30, 2 * 30)
104         if done and self.statistics:
105             if self.statisticsRecorder is None:
106                 self.statisticsRecorder = BarbellStatistics(self.env_name)
107             self.statisticsRecorder.save(self.current_epoch, self.total_reward)
108         return np.array(observation), reward, done
109
110     def render(self, mode='human', close=False):
111         if self.viewer is None:
112             self.viewer = BarbellViewer(self.viewport_width,
113                 self.viewport_height)
114         self.viewer.draw_objects(self.world.objects)
115         return self.viewer.render(return_rgb_array=(mode == 'rgb_array'))
```

Código-fonte 5.1: Implementação do Cartpole pelo Barbell

```
1  DOMAIN:
2    viewport_width: 600
3    fps_rate: 60
4    viewport_height: 400
5    ppm: 50
6    statistics: true
7
8  PARTS:
9    cart:
10       body_shape: box
11       body_type: dynamic
12       initial_position: [300, 200]
13       box_size: [60, 30]
14       color1: blue
15       z_index: 0
16   pole:
17       body_shape: box
18       body_type: dynamic
19       initial_position: [300, 400]
20       box_size: [5, 70]
```

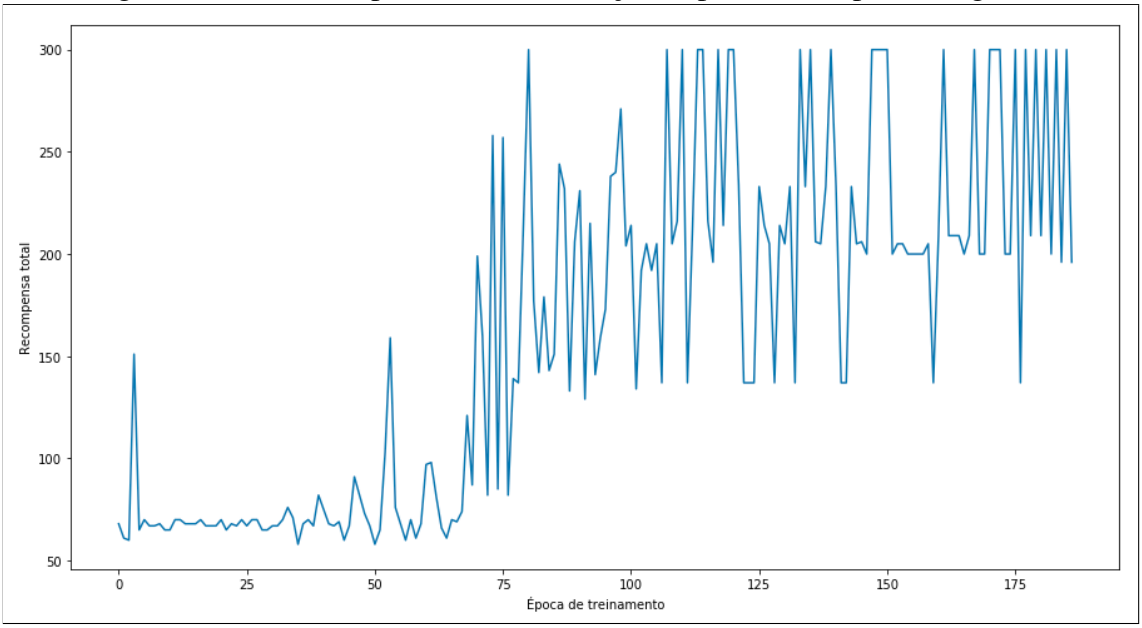
```
21        color1: black
22        angle: 0
23        z_index: 1
24    poletop:
25        body_shape: circle
26        body_type: dynamic
27        initial_position: [300, 450]
28        radius: 10
29        color1: white
30        color2: black
31        z_index: 3
32    floor:
33        body_shape: box
34        body_type: static
35        initial_position: [300, 10]
36        color1: white
37        box_size: [10, 10]
38
39
40  JOINTS:
41    - connects: [cart, pole]
42      type: revolute
43      local_anchor_b: [0, -70]
44    - connects: [pole, poletop]
45      type: revolute
46      local_anchor_a: [0, 60]
47    - connects: [floor, cart]
48      type: prismatic
49      anchor: [0, 0]
50      axis: [1, 0]
51      lower_translation: -3
52      upper_translation: 3
53
54  ENVIRONMENT:
55    gravity: [0, -10]
```

Código-fonte 5.2: Arquivo YAML de definição dos objetos do problema Cartpole

Nas primeiras linhas do código-fonte 5.2, informações sobre o domínio são definidas, sendo elas o tamanho da janela onde será renderizada a simulação, o fator de escala entre metros (unidade utilizada no simulador de física) e pixels (unidade utilizada pela ferramenta de renderização) e uma variável booleana chamada `statistics`, que

serve para ligar o salvamento automático da recompensa total recebida a cada época de treinamento. A seguir, os objetos e as juntas que os conectam são definidas: há uma junta do tipo `revolute` que conecta o carro com a haste, uma outra junta, também do tipo `revolute` que conecta a haste a um objeto (chamado de `poletop`) preso à sua extremidade, e há uma outra junta do tipo `prismatic`, que conecta o carro com um objeto estático, para simular o trilho. Há, além disso, uma chave chamada `z_index` para cada um dos objetos, que indica a ordem que eles são desenhados (para evitar que o carro sobreponha-se à haste, por exemplo).

Figura 5.3: curva de aprendizado da solução implementada pelo código 5.3



No código-fonte 5.1, apenas parte do método de inicialização (linhas 34 a 47), o método `observation()` (linhas 71 a 78), o método `done()` (linhas 80 a 85) e partes do método `reset()` (linhas 62 a 69) e `step()` devem ser implementados. De resto, o código é gerado automaticamente, incluindo ferramentas de controle que guardam informações a respeito da simulação, como a época atual (linha 59), qual é o instante da época atual (linha 102) e qual a recompensa total recebida até o momento (linha 101). Os objetos relacionados à renderização da simulação (método `render()`, linhas 110 a 114) e à gravação dos resultados obtidos (linhas 104 a 108) também são gerados automaticamente e, neste caso, não requerem alterações.

De todas as 114 linhas do `script` 5.1, apenas 51 foram escritas à mão, o resto sendo gerada automaticamente. Para fins de comparação, a implementação do problema Cartpole oferecida pelo catálogo do Gym pode ser encontrada no anexo A. No código, está explícito o cálculo das forças atuantes em cada um dos objetos em um determinado

instante de tempo, informação que é utilizada para fazer a transição entre os estados do sistema. No código 5.1, entretanto, tais cálculos são desnecessários, uma vez que Box2D, o motor de física atuante na simulação, realiza todos os cálculos necessários automaticamente.

O código 5.3 utiliza uma função de minimização para resolver o problema do Cartpole. Como os resultados são salvos automaticamente (gerando um arquivo CSV que contém, em cada linha, a época de treinamento e a recompensa total recebida ao final deste), uma curva de aprendizado pôde ser gerada (figura 5.3).

```
import gym
import numpy as np
import Cartpole  # NOQA
from scipy.optimize import minimize
x0 = np.zeros(4)
def run_cartpole(weights, render=False):
    env = gym.make('cartpole-v0')
    env.set_definition_file('cartpole_definition.yaml')
    env.reset()
    observation = np.zeros(4)
    done = False
    total_reward = 0
    while not done:
        if np.sum(weights * observation) <= 0:
            action = 0
        else:
            action = 1
        observation, reward, done = env.step(action)
        total_reward += reward
    return -1 * total_reward


result = minimize(run_cartpole, x0, method="Nelder-Mead")
```

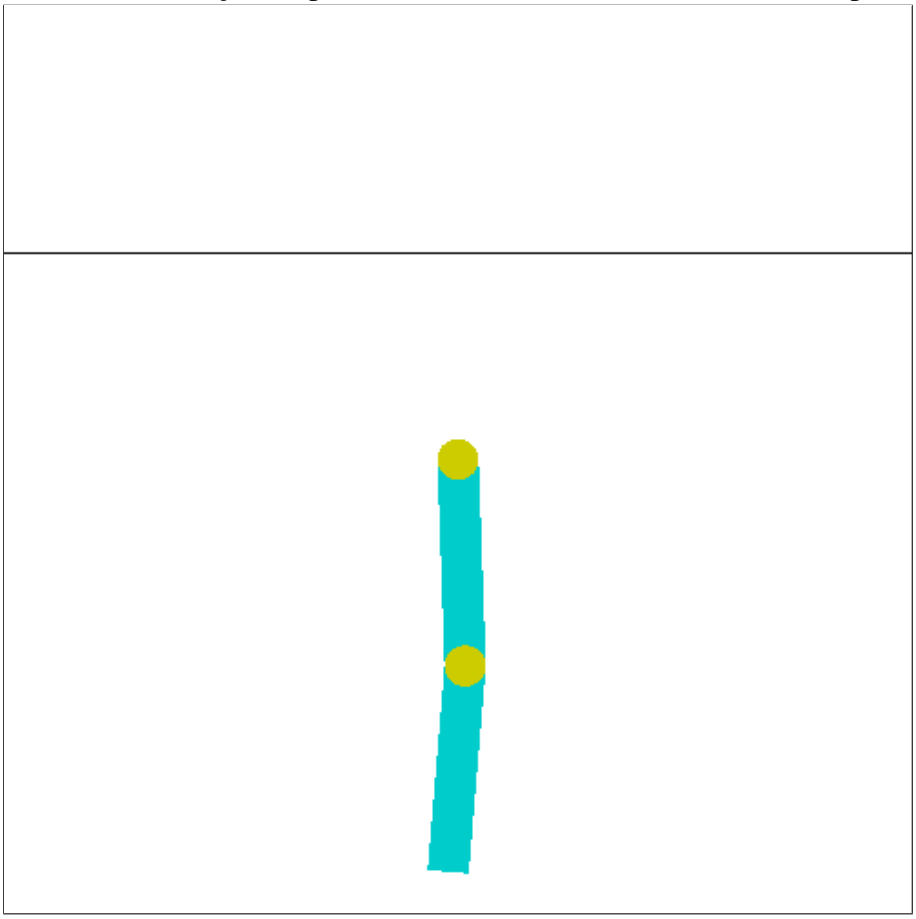
Código-fonte 5.3: Resolução do problema do Cartpole

5.2 Acrobot

O problema conhecido como AcroBot, inicialmente proposto em (SUTTON, 1996), consiste em um sistema formado por duas hastes A e B conectadas por suas extremidades, com uma das hastes tendo a sua outra extremidade fixada. A haste A, fixada, pode

rodar ao redor do seu ponto fixo, e o objetivo é fazer com que as duas hastes atinjam um ângulo próximo de 180 graus em relação à posição inicial das hastes (figura 5.4), sendo o episódio considerado terminado quando o objetivo é atingido ou quando a simulação chega no passo de número 5000.

Figura 5.4: Renderização do problema Acrobot, fornecido nativamente pelo Gym



As informações do ambiente que são passadas ao agente vêm na forma de um vetor de seis elementos, que são:

- Cosseno do ângulo da haste fixada (haste A);
- seno do ângulo da haste fixada (haste A);
- cosseno do ângulo da haste B, em relação à haste A;
- seno do ângulo da haste B, em relação à haste A;
- velocidade angular da haste A;
- velocidade angular da haste B.

Com base nestas informações, o agente deve decidir se toma uma entre duas ações, representadas pelo número -1 (girar a haste no sentido horário) e $+1$ (girar a haste no

sentido anti-horário). A recompensa é de -1 para cada passo de simulação, ou seja, episódios resolvidos mais rapidamente resultam em recompensas mais altas. O código que implementa a simulação está listado a seguir.

```
DOMAIN:
  viewport_width: 600
  viewport_height: 400
  fps_rate: 60
  ppm: 50

PARTS:
  circle1:
      body_shape: circle
      body_type: static
      initial_position: [300, 200]
      radius: 8
      color1: blue
      color2: black
      z_index: 1
  pole1:
      body_shape: box
      body_type: dynamic
      initial_position: [300, 160]
      box_size: [5, 50]
      color1: gray
      z_index: 0
  pole2:
      body_shape: box
      body_type: dynamic
      initial_position: [300, 100]
      box_size: [5, 50]
      color1: gray
      z_index: 0

JOINTS:
  - connects: [circle1, pole1]
    type: revolute
    local_anchor_b: [0, 45]
  - connects: [pole1, pole2]
    type: revolute
    local_anchor_a: [0, -45]
```

```
    local_anchor_b: [0, 45]


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

Código-fonte 5.4: Arquivo YAML que define o problema AcroBot

```
import gym
from barbell_environment import BarbellWorld, BarbellViewer, \
    BarbellContact, BarbellStatistics
from barbell_utils import parse_file
from math import sin, cos
import numpy as np

# CONSTANTS
FPS = 50  # desired FPS rate


class BarbellAcrobotContactDetector(BarbellContact):
    def __init__(self, env):
        BarbellContact.__init__(self)
        self.env = env

    def BeginContact(self, contact):
        pass

    def EndContact(self, contact):
        pass


class BarbellAcrobot(gym.Env):
    metadata = {
        'render.modes': ['human', 'rgb_array'],
        'video.frames_per_second': FPS
    }

    def __init__(self):
        self.viewer = None
        self.viewport_width = None
        self.viewport_height = None
        self.gravity = None
        self.drawlist = {}
```

```
36          self.partslist = {}
37          self.jointslist = []
38          self.name = 'barbellAcrobot-v0'
39          self.world = None
40          self.current_step = 0
41          self.current_epoch = 0
42          self.statistics = False
43          self.statisticsRecorder = None
44
45          high = np.array([1.0, 1.0, 1.0, 1.0, np.inf, np.inf])
46          low = -high
47
48          self.observation_space = gym.spaces.Box(low=low, high=high, dtype=np.float32)
49          self.action_space = gym.spaces.Discrete(3)
50
51          self.initialize_world()
52
53      def initialize_world(self):
54          if self.world is not None:
55              for body in self.world.objects:
56                  self.world.DestroyBody(self.world.objects[body])
57          self.world = BarbellWorld(gravity=self.gravity)
58          self.world.initialized_contact_detector = BarbellAcrobotContactDetector(self)
59          self.world.contactListener = self.world.initialized_contact_detector
60
61      def get_object(self, object_name):
62          return self.world.objects[object_name]
63
64      def get_joint(self, object_a, object_b):
65          return self.world.joints["%s_%s" % object_a, object_b]
66
67      def set_definition_file(self, filename):
68          parse_file(filename, self)
69
70      def reset(self):
71          self.current_step = 0
72          self.total_reward = 0
73          self.current_epoch += 1
74
75          self.initialize_world()
76          self.world.create_objects(self.partslist)
```

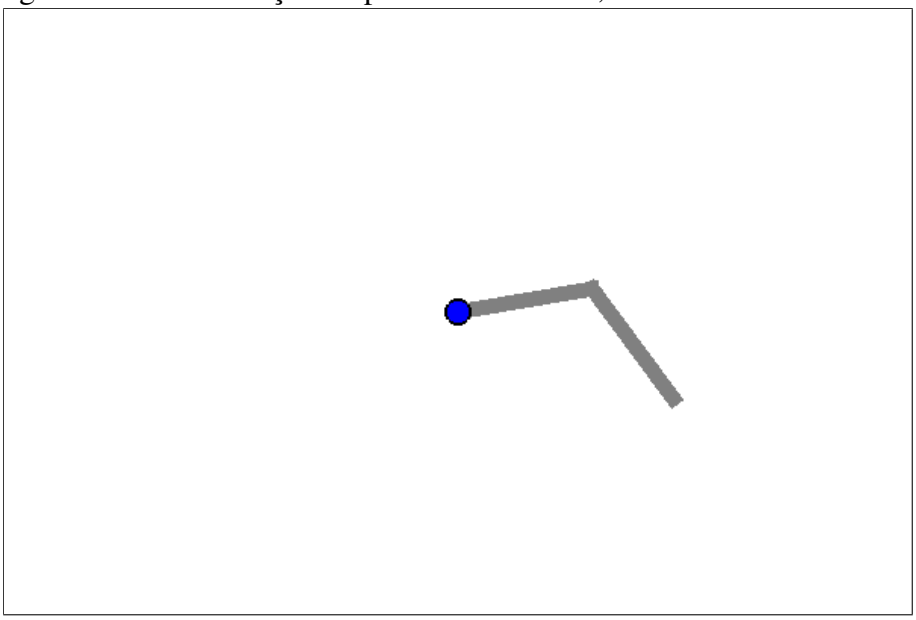
```
77          self.world.create_joints(self.jointslist)
78
79      def observation(self):
80          angle1 = self.get_object('pole1').angle
81          angle2 = self.get_object('pole2').angle - angle1
82          lv1 = self.get_object('pole1').angularVelocity
83          lv2 = self.get_object('pole2').angularVelocity
84          obs = [cos(angle1), sin(angle1), cos(angle2), sin(angle2), lv1, lv2]
85          return np.array(obs)
86
87      def reward(self):
88          return -1
89
90      def done(self, observation):
91          angle1 = self.get_object('pole1').angle
92          angle2 = self.get_object('pole2').angle - angle1
93          return self.current_step >= 2000 or bool(-cos(angle1) - \
94            cos(angle1 + angle2) > 1.)
95
96      def step(self, action):
97          self.world.apply_force('rotate', 'pole1', action * 1)
98
99          self.current_step += 1
100         reward = self.reward()
101         observation = self.observation()
102         done = self.done(observation)
103         self.world.Step(1.0 / FPS, 6 * 30, 2 * 30)
104         if done and self.statistics:
105             if self.statisticsRecorder is None:
106                 self.statisticsRecorder = BarbellStatistics(self.env_name)
107             self.statisticsRecorder.save(self.current_epoch, self.total_reward)
108         return observation, reward, done
109
110     def render(self, mode='human', close=False):
111         if self.viewer is None:
112                     self.viewer = BarbellViewer(viewport_width=self.viewport_width,
113                     viewport_height=self.viewport_height)
114
115         self.viewer.draw_objects(self.world.objects)
116         return self.viewer.render(return_rgb_array=(mode == 'rgb_array'))
```

Código-fonte 5.5: *Script* Python que define o problema AcroBot

No código 5.4, são definidos três objetos: um círculo fixo, formado por um corpo do tipo `static`, ou seja, que não sofre a ação das forças físicas da simulação, a haste A (chamada no código de `pole1`), conectada ao círculo, e a haste B (chamada no código de `pole2`), conectada à haste A. As juntas do tipo `revolute` permitem que dois corpos sejam conectados em um ponto e rotacionem usando o ponto como eixo, sem que haja colisão entre os objetos.

No código 5.5, os métodos `observation()`, `reward()` e `done()` (linhas 79 a 94) devem ser implementados integralmente, enquanto a linha 97 no método `step()` e as linhas 75 a 78 no método `reset()` devem ser adicionadas para completá-los. A figura 5.5 mostra o resultado da renderização da simulação.

Figura 5.5: Renderização do problema Acrobot, desenvolvido no Barbell

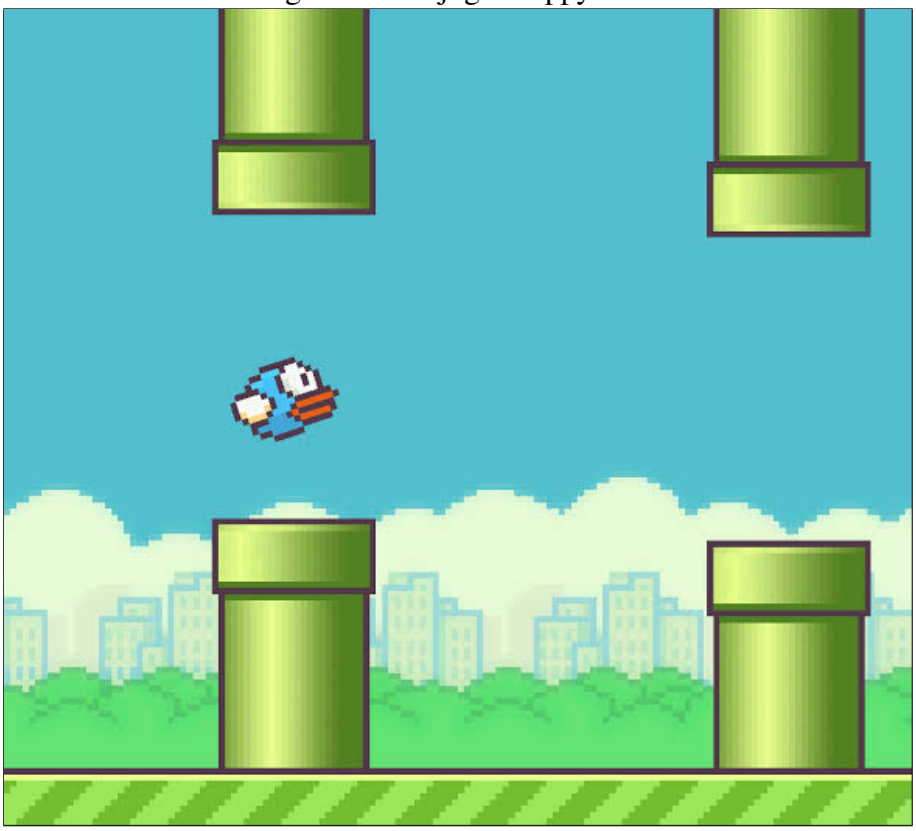


5.3 Flappy Bird

O jogo Flappy Bird, feito para dispositivos móveis no começo da década, por se tratar de um jogo bastante simples, mas que, ao mesmo tempo, requer habilidade, já foi alvo de várias pesquisas no campo de aprendizado por reforço (PIPER; BHOUNSULE; CASTILLO-VILLAR, 2017; APPIAH; VARE, 2018; CHEN, 2015). Nele, um pássaro, sempre navegando para a frente, deve passar por aberturas entre canos, e o jogador decide quando é necessário impulsionar o pássaro para cima (figura 5.6)

A formulação do jogo como um problema de aprendizado por reforço é bastante

Figura 5.6: O jogo Flappy Bird



simples: o agente deve controlar o pássaro, com o episódio de treinamento chegando ao fim toda vez que o pássaro colide com algum cano ou com o chão. A recompensa é proporcional ao tempo em que o pássaro consegue manter-se vivo — como no exemplo do Cartpole, a recompensa é de $+1$ para cada passo da simulação. Baseado no estado do ambiente (mais sobre isso a seguir), o agente deve decidir se toma a única ação possível (impulsionar o pássaro para cima) ou não tomar ação alguma. A implementação do problema é bastante simples e os arquivos Python e YAML que o definem estão listados a seguir:

```
DOMAIN:
  viewport_width: 1000
  viewport_height: 500
  ppm: 50
  statistics: True


PARTS:
  bird:
      body_type: dynamic
      body_shape: box
```

```
        initial_position: [512, 384]
        color1: blue
        color2: black
        box_size: [20, 10]
    floor:
        body_type: static
        body_shape: box
        initial_position: [600, -2]
        color1: white
        box_size: [600, 1]
    ceiling:
        body_type: static
        body_shape: box
        initial_position: [600, 1202]
        color1: white
        box_size: [600, 1]
    pipe1_top:
        body_type: static
        body_shape: box
        color1: dark_olive_green
        box_size: [30, 150]
        initial_position: [1030, 350]
    pipe1_bottom:
        body_type: static
        body_shape: box
        color1: dark_olive_green
        box_size: [30,50]
        initial_position: [1030, 50]
        z_index: 1
    pipe2_top:
        body_type: static
        body_shape: box
        color1: dark_olive_green
        box_size: [30, 100]
        initial_position: [1030, 400]
    pipe2_bottom:
        body_type: static
        body_shape: box
        color1: dark_olive_green
        box_size: [30,100]
        initial_position: [1030, 100]
```

```
      z_index: 1
  pipe3_top:
      body_type: static
      body_shape: box
      color1: dark_olive_green
      box_size: [30, 50]
      initial_position: [1030, 450]
  pipe3_bottom:
      body_type: static
      body_shape: box
      color1: dark_olive_green
      box_size: [30, 150]
      initial_position: [1030, 150]
  pipe4_bottom:
      body_type: static
      body_shape: box
      color1: dark_olive_green
      box_size: [30, 200]
      initial_position: [1030, 200]
  pipe4_top:
      body_type: static
      body_shape: box
      color1: dark_olive_green
      box_size: [30, 2]
      initial_position: [1030, 498]
  pipe5_bottom:
     body_type: static
     body_shape: box
     color1: dark_olive_green
     box_size: [30, 2]
     initial_position: [1030, 0]
  pipe5_top:
     body_type: static
     body_shape: box
     color1: dark_olive_green
     box_size: [30, 200]
     initial_position: [1030, 300]
```

Código-fonte 5.6: Arquivo YAML que define o cenário do Flappy Bird

```
import gym
import random
from barbell_environment import BarbellWorld, BarbellViewer, BarbellContact, BarbellSt
```

```
4  from barbell_utils import parse_file
5
6
7  # CONSTANTS
8  FPS = 50  # desired FPS rate
9
10
11 class FlappybirdContactDetector(BarbellContact):
12     def __init__(self, env):
13         BarbellContact.__init__(self)
14         self.env = env
15
16     def BeginContact(self, contact):
17         self.env.collided = True
18
19     def EndContact(self, contact):
20         self.env.collided = False
21
22
23 class Flappybird(gym.Env):
24     metadata = {
25         'render.modes': ['human', 'rgb_array'],
26         'video.frames_per_second': FPS
27     }
28
29     def __init__(self):
30         self.env_name = 'flappybird-v0'
31         self.viewer = None
32         self.drawlist = {}
33         self.viewport_width = None
34         self.viewport_height = None
35         self.gravity = None
36         self.partslist = {}
37         self.jointslist = []
38         self.world = None
39         self.collided = False
40         self.current_step = 0
41         self.current_epoch = 0
42         self.statistics = False
43         self.statisticsRecorder = None
44
```

```
45          self.initialize_world({})
46
47      def initialize_world(self, objects):
48          if self.world is not None:
49              for body in self.world.objects:
50                  self.world.DestroyBody(self.world.objects[body])
51          self.world = BarbellWorld(gravity=self.gravity)
52          self.world.initialized_contact_detector = FlappybirdContactDetector(self)
53          self.world.contactListener = self.world.initialized_contact_detector
54          self.world.create_objects(objects)
55
56      def get_object(self, object_name):
57          return self.world.objects[object_name]
58
59      def get_joint(self, object_a, object_b):
60          return self.world.joints["%s_%s" % object_a, object_b]
61
62      def set_definition_file(self, filename):
63          parse_file(filename, self)
64
65      def reset(self):
66          self.current_step = 0
67          self.total_reward = 0
68          self.current_epoch += 1
69          objects = {}
70          objects['bird'] = self.partslist['bird']
71          objects['floor'] = self.partslist['floor']
72          objects['ceiling'] = self.partslist['ceiling']
73          for i in range(4):
74              pipe = random.randint(1, 4)
75              pipe_top = dict(self.partslist['pipe%d_top' % pipe])
76              pipe_bottom = dict(self.partslist['pipe%d_bottom' % pipe])
77              pipe_top['initial_position'] = (pipe_top['initial_position'][0] \
78                    + (i * 250), pipe_top['initial_position'][1])
79              pipe_bottom['initial_position'] = \
80                (pipe_bottom['initial_position'][0] \
81                + (i * 250), pipe_bottom['initial_position'][1])
82              objects['pipe_%d_top' % i] = pipe_top
83              objects['pipe_%d_bottom' % i] = pipe_bottom
84          self.initialize_world(objects)
85
```

```
86      def observation(self):
87          pass
88
89      def reward(self):
90          if self.done():
91              return 0
92          else:
93              return 1
94
95      def done(self):
96          if self.collided or self.current_step >= 2000:
97              return True
98          else:
99              return False
100
101     def step(self, action):
102         if random.randint(1, 20) == 2:
103             self.world.apply_force('local', 'bird', (0, 40))
104
105         to_delete = []
106         for key in self.world.objects:
107             if key.startswith("pipe"):
108                 pipe_object = self.get_object(key)
109                 if pipe_object.position[0] * self.ppm < -30:
110                     to_delete.append(key)
111
112         if len(to_delete) > 0:
113             self.world.destroy_object(to_delete[0])
114             self.world.destroy_object(to_delete[1])
115             new_pipe = random.randint(1, 5)
116             new_pipe_top = dict(self.partslist['pipe%d_top' % new_pipe])
117             new_pipe_bottom = dict(self.partslist['pipe%d_bottom' % new_pipe])
118             self.world.create_object(to_delete[0], new_pipe_top)
119             self.world.create_object(to_delete[1], new_pipe_bottom)
120
121         reward = self.reward()
122         self.total_reward += reward
123         observation = self.observation()
124         done = self.done()
125         self.current_step += 1
126         self.world.Step(1.0 / FPS, 6 * 30, 2 * 30)
```

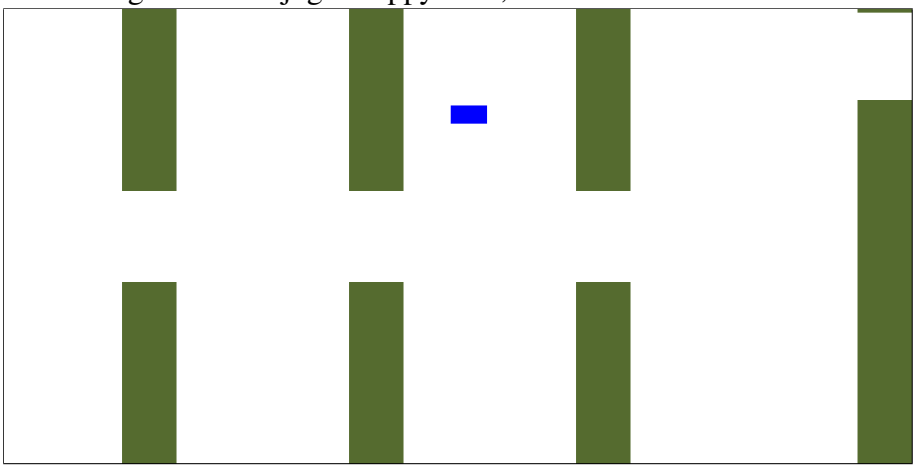
```
127         if done and self.statistics:
128             if self.statisticsRecorder is None:
129                 self.statisticsRecorder = BarbellStatistics(self.env_name)
130             self.statisticsRecorder.save(self.current_epoch, self.total_reward)
131         return observation, reward, done
132 
133     def render(self, mode='human', close=False):
134         if self.viewer is None:
135                     self.viewer = BarbellViewer(
136                       viewport_width=self.viewport_width,
137                       viewport_height=self.viewport_height)
138         self.viewer.move_camera(self.world.objects, (-2, 0))
139         self.viewer.move_camera({
140             'b': self.world.objects['bird'],
141             'floor': self.world.objects['floor'],
142             'ceiling': self.world.objects['ceiling']}, (2, 0))
143         self.viewer.draw_objects(self.world.objects)
144         return self.viewer.render(return_rgb_array=(mode == 'rgb_array'))
```

Código-fonte 5.7: *Script* Python que define o cenário do Flappy Bird

A lógica por trás do algoritmo listado nos códigos 5.6 e 5.7 é bastante simples: define-se o pássaro, dois objetos, `floor` e `ceiling` que detectam se o pássaro saiu da tela e canos com aberturas de várias alturas diferentes. O pássaro e os objetos `floor` e `ceiling` estão presentes na simulação a todo momento. Os canos, por sua vez, movem-se para a esquerda da tela e, cada vez que um deles ultrapassa a extremidade esquerda desta, um outro é criado na extremidade direita, em uma espécie de esteira. Os objetos que definem os canos ficam salvos na propriedade `objectslist` e são criados de acordo com a lógica acima (linhas 105 a 119). Além da implementação desta lógica, os métodos `observation()`, `reward()` e `done()` (linhas 86 a 131) e a lógica que detecta colisões (linha 17) foram implementadas manualmente. O resultado pode ser visto na figura 5.7.

O método `observation()`, deixado intencionalmente em branco na simulação, poderia retornar, por exemplo, um vetor descrevendo todos os canos visíveis na tela e mais informações do pássaro, por exemplo. Entretanto, o método de representação do estado atual da simulação escolhido para este exemplo foi uma matriz de tamanho $W \times H \times 3$, onde W e H são a largura e a altura da tela, respectivamente. A matriz traz informações sobre os valores RGB de todos os pixels da tela, e pode ser informada a uma rede neural que, a partir das informações da tela, decide qual a próxima ação a ser executada. Tal ação

Figura 5.7: O jogo Flappy Bird, no ambiente do Barbell



é possível trocando o modo de renderização de `'human'` para `'rgb_array'`, como na linha 13 do código 5.8.

```
1  import random
2  import gym
3  import FlappyBird # noqa
4
5  env = gym.make('flappybird-v0')
6  env.set_definition_file('flappybird_definition.yaml')
7  env.reset()
8
9  while True:
10     observation, reward, done,  = env.step(int(random.randint(0, 1)))
11     if done:
12         env.reset()
13     env.render(mode='rgb_array')
```

Código-fonte 5.8: Exemplo de uso do modo `'rgb_array'`

6 CONCLUSÃO E TRABALHOS FUTUROS

Neste trabalho foi visto que não é uma tarefa fácil modelar um problema de aprendizado por reforço e criar um ambiente que simule o problema é ainda mais desafiador. Incontáveis são as ferramentas que podem ser utilizadas para construir uma simulação, e escolher a ferramenta certa, dada a amplitude de aplicações diferentes que determinadas ferramentas podem ter, é uma tarefa que exige atenção. Este trabalho é uma tentativa não só de contornar algumas das dificuldades encontradas na hora de escolher a ferramenta certa para a construção de uma simulação de aprendizado por reforço, como de oferecer alguns atalhos para aqueles que pretendem construir ferramentas do tipo.

Os atalhos que o Barbell fornece visam um código mais claro, limpo e padronizado, e nisto esta ferramenta cumpre muito bem o seu papel. Todos os cenários produzidos através do Barbell são completamente compatíveis com a padronização de simulações de aprendizado por reforço proposta pelo Gym, e a geração de código automática para a produção destas simulações remove preocupações não relacionadas à formulação do problema em si, fazendo com que os esforços sejam dirigidos apenas àquilo que interessa: ao agente de aprendizado por reforço e ao problema que ele tenta resolver.

Uma das limitações do Barbell, entretanto, é o fato de que apenas um único motor de física é compatível com sua API. No futuro, será possível escrever definições de objetos em YAML e trocar o motor de física com apenas um comando, sem que a linguagem compreendida pelo Barbell precise ser alterada. Isto adicionará mais uma camada de abstração no processo de construção de cenários e permitirá que uma mesma solução seja rapidamente aplicada a um motor de física diferente.

ANEXO A: CÓDIGO DA IMPLEMENTAÇÃO DO CARTPOLE PELO GYM

```
"""
Classic cart-pole system implemented by Rich Sutton et al.
Copied from http://incompleteideas.net/sutton/book/code/pole.c
permalink: https://perma.cc/C9ZM-652R
"""

import math
import gym
from gym import spaces, logger
from gym.utils import seeding
import numpy as np

class CartPoleEnv(gym.Env):
    """
    Description:
        A pole is attached by an un-actuated joint to a cart, which moves
        along a frictionless track. The pendulum starts upright, and the goal
        is to prevent it from falling over by
        increasing and reducing the cart's velocity.
    Source:
        This environment corresponds to the version of the cart-pole problem
        described by Barto, Sutton, and Anderson
    Observation:
        Type: Box(4)
        Num         Observation                Min         Max
        0         Cart Position             -4.8           4.8
        1         Cart Velocity             -Inf           Inf
        2         Pole Angle                 -24 deg       24 deg
        3         Pole Velocity At Tip      -Inf           Inf

    Actions:
        Type: Discrete(2)
        Num         Action
        0         Push cart to the left
        1         Push cart to the right

        Note: The amount the velocity that is reduced or increased is not fixed;
        it depends on the angle the pole is pointing. This is because the
        center of gravity of the pole increases the amount of energy needed
```

```
40          to move the cart underneath it
41      Reward:
42          Reward is 1 for every step taken, including the termination step
43      Starting State:
44          All observations are assigned a uniform random value in [-0.05..0.05]
45      Episode Termination:
46          Pole Angle is more than 12 degrees
47          Cart Position is more than 2.4
48            (center of the cart reaches the edge of the display)
49          Episode length is greater than 200
50          Solved Requirements
51          Considered solved when the average reward is greater than or equal
52          to 195.0 over 100 consecutive trials.
53      """
54
55      metadata = {
56          'render.modes': ['human', 'rgb_array'],
57          'video.frames_per_second' : 50
58      }
59
60      def __init__(self):
61          self.gravity = 9.8
62          self.masscart = 1.0
63          self.masspole = 0.1
64          self.total_mass = (self.masspole + self.masscart)
65          self.length = 0.5 # actually half the pole's length
66          self.polemass_length = (self.masspole * self.length)
67          self.force_mag = 10.0
68          self.tau = 0.02  # seconds between state updates
69          self.kinematics_integrator = 'euler'
70
71          # Angle at which to fail the episode
72          self.theta_threshold_radians = 12 * 2 * math.pi / 360
73          self.x_threshold = 2.4
74
75          high = np.array([
76              self.x_threshold * 2,
77              np.finfo(np.float32).max,
78              self.theta_threshold_radians * 2,
79              np.finfo(np.float32).max])
80
```

```
81          self.action_space = spaces.Discrete(2)
82          self.observation_space = spaces.Box(-high, high, dtype=np.float32)
83
84          self.seed()
85          self.viewer = None
86          self.state = None
87
88          self.steps_beyond_done = None
89
90      def seed(self, seed=None):
91          self.np_random, seed = seeding.np_random(seed)
92          return [seed]
93
94      def step(self, action):
95          assert self.action_space.contains(action),
96                  "%r (%s) invalid"%(action, type(action))
97          state = self.state
98          x, x_dot, theta, theta_dot = state
99          force = self.force_mag if action==1 else -self.force_mag
100         costheta = math.cos(theta)
101         sintheta = math.sin(theta)
102         temp = (force + self.polemass_length * theta_dot * theta_dot \
103                               * sintheta) / self.total_mass
104         thetaacc = (self.gravity * sintheta - costheta* temp) /
105           (self.length * (4.0/3.0 - self.masspole *  \
106                           costheta * costheta / self.total_mass))
107         xacc  = temp - self.polemass_length * thetaacc  \
108                     * costheta / self.total_mass
109         if self.kinematics_integrator == 'euler':
110             x  = x + self.tau * x_dot
111             x_dot = x_dot + self.tau * xacc
112             theta = theta + self.tau * theta_dot
113             theta_dot = theta_dot + self.tau * thetaacc
114         else: # semi-implicit euler
115             x_dot = x_dot + self.tau * xacc
116             x  = x + self.tau * x_dot
117             theta_dot = theta_dot + self.tau * thetaacc
118             theta = theta + self.tau * theta_dot
119         self.state = (x,x_dot,theta,theta_dot)
120         done =  x < -self.x_threshold \
121                 or x > self.x_threshold \
```

```
122                 or theta < -self.theta_threshold_radians \
123                 or theta > self.theta_threshold_radians
124         done = bool(done)
125 
126         if not done:
127             reward = 1.0
128         elif self.steps_beyond_done is None:
129             # Pole just fell!
130             self.steps_beyond_done = 0
131             reward = 1.0
132         else:
133             if self.steps_beyond_done == 0:
134                 logger.warn("You are calling 'step()' even though \
135                 this environment \
136                  has already returned done = True. \
137                  You should always call 'reset()' \
138                   once you receive 'done = True' -- \
139                   any further steps are undefined behavior.")
140             self.steps_beyond_done += 1
141             reward = 0.0
142 
143         return np.array(self.state), reward, done, {}
144 
145     def reset(self):
146         self.state = self.np_random.uniform(low=-0.05, high=0.05, size=(4,))
147         self.steps_beyond_done = None
148         return np.array(self.state)
149 
150     def render(self, mode='human'):
151         screen_width = 600
152         screen_height = 400
153 
154         world_width = self.x_threshold*2
155         scale = screen_width/world_width
156         carty = 100 # TOP OF CART
157         polewidth = 10.0
158         polelen = scale * (2 * self.length)
159         cartwidth = 50.0
160         cartheight = 30.0
161 
162         if self.viewer is None:
```

```
163             from gym.envs.classic_control import rendering
164             self.viewer = rendering.Viewer(screen_width, screen_height)
165             l,r,t,b = -cartwidth/2, cartwidth/2, cartheight/2, -cartheight/2
166             axleoffset =cartheight/4.0
167             cart = rendering.FilledPolygon([(l,b), (l,t), (r,t), (r,b)])
168             self.carttrans = rendering.Transform()
169             cart.add_attr(self.carttrans)
170             self.viewer.add_geom(cart)
171             l,r,t,b = -polewidth/2,polewidth/2,polelen-polewidth/2,-polewidth/2
172             pole = rendering.FilledPolygon([(l,b), (l,t), (r,t), (r,b)])
173             pole.set_color(.8,.6,.4)
174             self.poletrans = rendering.Transform(translation=(0, axleoffset))
175             pole.add_attr(self.poletrans)
176             pole.add_attr(self.carttrans)
177             self.viewer.add_geom(pole)
178             self.axle = rendering.make_circle(polewidth/2)
179             self.axle.add_attr(self.poletrans)
180             self.axle.add_attr(self.carttrans)
181             self.axle.set_color(.5,.5,.8)
182             self.viewer.add_geom(self.axle)
183             self.track = rendering.Line((0,carty), (screen_width,carty))
184             self.track.set_color(0,0,0)
185             self.viewer.add_geom(self.track)
186
187             self._pole_geom = pole
188
189         if self.state is None: return None
190
191         # Edit the pole polygon vertex
192         pole = self._pole_geom
193         l,r,t,b = -polewidth/2,polewidth/2,polelen-polewidth/2,-polewidth/2
194         pole.v = [(l,b), (l,t), (r,t), (r,b)]
195
196         x = self.state
197         cartx = x[0]*scale+screen_width/2.0 # MIDDLE OF CART
198         self.carttrans.set_translation(cartx, carty)
199         self.poletrans.set_rotation(-x[2])
200
201         return self.viewer.render(return_rgb_array = mode=='rgb_array')
202
203     def close(self):
```

```
204         if self.viewer:
205             self.viewer.close()
206             self.viewer = None
```

Código-fonte 6.1: Implementação do Cartpole disponibilizada pela biblioteca do Gym

ANEXO B: CÓDIGO DA IMPLEMENTAÇÃO DO ACROBOT PELO GYM

```
"""classic Acrobot task"""
import numpy as np
from numpy import sin, cos, pi

from gym import core, spaces
from gym.utils import seeding

__copyright__ = "Copyright 2013, RLPy http://acl.mit.edu/RLPy"
__credits__ = ["Alborz Geramifard", "Robert H. Klein", "Christoph Dann",
               "William Dabney", "Jonathan P. How"]
__license__ = "BSD 3-Clause"
__author__ = "Christoph Dann <cdann@cdann.de>"

# SOURCE:
# https://github.com/rlpy/rlpy/blob/master/rlpy/Domains/Acrobot.py

class AcrobotEnv(core.Env):

    """
    Acrobot is a 2-link pendulum with only the second joint actuated.
    Initially, both links point downwards. The goal is to swing the
    end-effector at a height at least the length of one link above the base.
    Both links can swing freely and can pass by each other, i.e., they don't
    collide when they have the same angle.
    **STATE:**
    The state consists of the sin() and cos() of the two rotational joint
    angles and the joint angular velocities :
    [cos(theta1) sin(theta1) cos(theta2) sin(theta2) thetaDot1 thetaDot2].
    For the first link, an angle of 0 corresponds to the link pointing downwards.
    The angle of the second link is relative to the angle of the first link.
    An angle of 0 corresponds to having the same angle between the two links.
    A state of [1, 0, 1, 0, ..., ...] means that both links point downwards.
    **ACTIONS:**
    The action is either applying +1, 0 or -1 torque on the joint between
    the two pendulum links.
    .. note::
        The dynamics equations were missing some terms in the NIPS paper which
        are present in the book. R. Sutton confirmed in personal correspondence
        that the experimental results shown in the paper and the book were
```

```
40          generated with the equations shown in the book.
41          However, there is the option to run the domain with the paper equations
42          by setting book_or_nips = 'nips'
43      **REFERENCE:**
44      .. seealso::
45          R. Sutton: Generalization in Reinforcement Learning:
46          Successful Examples Using Sparse Coarse Coding (NIPS 1996)
47      .. seealso::
48          R. Sutton and A. G. Barto:
49          Reinforcement learning: An introduction.
50          Cambridge: MIT press, 1998.
51      .. warning::
52          This version of the domain uses the Runge-Kutta method for integrating
53          the system dynamics and is more realistic, but also considerably harder
54          than the original version which employs Euler integration,
55          see the AcrobotLegacy class.
56      """
57
58      metadata = {
59          'render.modes': ['human', 'rgb_array'],
60          'video.frames_per_second' : 15
61      }
62
63      dt = .2
64
65      LINK_LENGTH_1 = 1.  # [m]
66      LINK_LENGTH_2 = 1.  # [m]
67      LINK_MASS_1 = 1.  #: [kg] mass of link 1
68      LINK_MASS_2 = 1.  #: [kg] mass of link 2
69      LINK_COM_POS_1 = 0.5  #: [m] position of the center of mass of link 1
70      LINK_COM_POS_2 = 0.5  #: [m] position of the center of mass of link 2
71      LINK_MOI = 1.  #: moments of inertia for both links
72
73      MAX_VEL_1 = 4 * pi
74      MAX_VEL_2 = 9 * pi
75
76      AVAIL_TORQUE = [-1., 0., +1]
77
78      torque_noise_max = 0.
79
80      #: use dynamics equations from the nips paper or the book
```

```
81      book_or_nips = "book"
82      action_arrow = None
83      domain_fig = None
84      actions_num = 3
85
86      def __init__(self):
87          self.viewer = None
88          high = np.array([1.0, 1.0, 1.0, 1.0, self.MAX_VEL_1, self.MAX_VEL_2])
89          low = -high
90          self.observation_space = spaces.Box(low=low, high=high, dtype=np.float32)
91          self.action_space = spaces.Discrete(3)
92          self.state = None
93          self.seed()
94
95      def seed(self, seed=None):
96          self.np_random, seed = seeding.np_random(seed)
97          return [seed]
98
99      def reset(self):
100         self.state = self.np_random.uniform(low=-0.1, high=0.1, size=(4,))
101         return self._get_ob()
102
103     def step(self, a):
104         s = self.state
105         torque = self.AVAIL_TORQUE[a]
106
107         # Add noise to the force action
108         if self.torque_noise_max > 0:
109             torque += self.np_random.uniform(-self.torque_noise_max,
110                                              self.torque_noise_max)
111
112         # Now, augment the state with our force action so it can be passed to
113         # _dsdt
114         s_augmented = np.append(s, torque)
115
116         ns = rk4(self._dsdt, s_augmented, [0, self.dt])
117         # only care about final timestep of integration returned by integrator
118         ns = ns[-1]
119         ns = ns[:4]  # omit action
120         # ODEINT IS TOO SLOW!
121         # ns_continuous = integrate.odeint(self._dsdt,
```

```
122         #self.s_continuous, [0, self.dt])
123         # self.s_continuous = ns_continuous[-1] # We only care about the state
124         # at the ''final timestep'', self.dt
125 
126         ns[0] = wrap(ns[0], -pi, pi)
127         ns[1] = wrap(ns[1], -pi, pi)
128         ns[2] = bound(ns[2], -self.MAX_VEL_1, self.MAX_VEL_1)
129         ns[3] = bound(ns[3], -self.MAX_VEL_2, self.MAX_VEL_2)
130         self.state = ns
131         terminal = self._terminal()
132         reward = -1. if not terminal else 0.
133         return (self._get_ob(), reward, terminal, {})
134 
135     def _get_ob(self):
136         s = self.state
137         return np.array([cos(s[0]), sin(s[0]), cos(s[1]), sin(s[1]), s[2], s[3]])
138 
139     def _terminal(self):
140         s = self.state
141         return bool(-cos(s[0]) - cos(s[1] + s[0]) > 1.)
142 
143     def _dsdt(self, s_augmented, t):
144         m1 = self.LINK_MASS_1
145         m2 = self.LINK_MASS_2
146         l1 = self.LINK_LENGTH_1
147         lc1 = self.LINK_COM_POS_1
148         lc2 = self.LINK_COM_POS_2
149         I1 = self.LINK_MOI
150         I2 = self.LINK_MOI
151         g = 9.8
152         a = s_augmented[-1]
153         s = s_augmented[:-1]
154         theta1 = s[0]
155         theta2 = s[1]
156         dtheta1 = s[2]
157         dtheta2 = s[3]
158         d1 = m1 * lc1 ** 2 + m2 * \
159             (l1 ** 2 + lc2 ** 2 + 2 * l1 * lc2 * cos(theta2)) + I1 + I2
160         d2 = m2 * (lc2 ** 2 + l1 * lc2 * cos(theta2)) + I2
161         phi2 = m2 * lc2 * g * cos(theta1 + theta2 - pi / 2.)
162         phi1 = - m2 * l1 * lc2 * dtheta2 ** 2 * sin(theta2) \
```

```
                - 2 * m2 * l1 * lc2 * dtheta2 * dtheta1 * sin(theta2)  \
            + (m1 * lc1 + m2 * l1) * g * cos(theta1 - pi / 2) + phi2
        if self.book_or_nips == "nips":
            # the following line is consistent with the description in the
            # paper
            ddtheta2 = (a + d2 / d1 * phi1 - phi2) / \
                (m2 * lc2 ** 2 + I2 - d2 ** 2 / d1)
        else:
            # the following line is consistent with the java implementation and the
            # book
            ddtheta2 = (a + d2 / d1 * phi1 - m2 * l1 * lc2 * dtheta1 ** 2  \
            * sin(theta2) - phi2) \
                / (m2 * lc2 ** 2 + I2 - d2 ** 2 / d1)
        ddtheta1 = -(d2 * ddtheta2 + phi1) / d1
        return (dtheta1, dtheta2, ddtheta1, ddtheta2, 0.)

    def render(self, mode='human'):
        from gym.envs.classic_control import rendering

        s = self.state

        if self.viewer is None:
            self.viewer = rendering.Viewer(500,500)
            bound = self.LINK_LENGTH_1 + self.LINK_LENGTH_2 + 0.2  # 2.2 for default
            self.viewer.set_bounds(-bound,bound,-bound,bound)

        if s is None: return None

        p1 = [-self.LINK_LENGTH_1 *
              cos(s[0]), self.LINK_LENGTH_1 * sin(s[0])]

        p2 = [p1[0] - self.LINK_LENGTH_2 * cos(s[0] + s[1]),
              p1[1] + self.LINK_LENGTH_2 * sin(s[0] + s[1])]

        xys = np.array([[0,0], p1, p2])[:,::-1]
        thetas = [s[0]- pi/2, s[0]+s[1]-pi/2]
        link_lengths = [self.LINK_LENGTH_1, self.LINK_LENGTH_2]

        self.viewer.draw_line((-2.2, 1), (2.2, 1))
        for ((x,y),th,llen) in zip(xys, thetas, link_lengths):
            l,r,t,b = 0, llen, .1, -.1
```

```
204                 jtransform = rendering.Transform(rotation=th, translation=(x,y))
205                 link = self.viewer.draw_polygon([(l,b), (l,t), (r,t), (r,b)])
206                 link.add_attr(jtransform)
207                 link.set_color(0,.8, .8)
208                 circ = self.viewer.draw_circle(.1)
209                 circ.set_color(.8, .8, 0)
210                 circ.add_attr(jtransform)
211 
212             return self.viewer.render(return_rgb_array = mode=='rgb_array')
213 
214     def close(self):
215         if self.viewer:
216             self.viewer.close()
217             self.viewer = None
218 
219 def wrap(x, m, M):
220     """
221     :param x: a scalar
222     :param m: minimum possible value in range
223     :param M: maximum possible value in range
224     Wraps ``x`` so m <= x <= M; but unlike ``bound()`` which
225     truncates, ``wrap()`` wraps x around the coordinate system defined by m,M.\n
226     For example, m = -180, M = 180 (degrees), x = 360 --> returns 0.
227     """
228     diff = M - m
229     while x > M:
230         x = x - diff
231     while x < m:
232         x = x + diff
233     return x
234 
235 def bound(x, m, M=None):
236     """
237     :param x: scalar
238     Either have m as scalar, so bound(x,m,M) which returns m <= x <= M *OR*
239     have m as length 2 vector, bound(x,m, <IGNORED>) returns m[0] <= x <= m[1].
240     """
241     if M is None:
242         M = m[1]
243         m = m[0]
244     # bound x between min (m) and Max (M)
```

```
245     return min(max(x, m), M)
246
247
248 def rk4(derivs, y0, t, *args, **kwargs):
249     """
250     Integrate 1D or ND system of ODEs using 4-th order Runge-Kutta.
251     This is a toy implementation which may be useful if you find
252     yourself stranded on a system w/o scipy.  Otherwise use
253     :func:`scipy.integrate`.
254     *y0*
255         initial state vector
256     *t*
257         sample times
258     *derivs*
259         returns the derivative of the system and has the
260         signature ``dy = derivs(yi, ti)``
261     *args*
262         additional arguments passed to the derivative function
263     *kwargs*
264         additional keyword arguments passed to the derivative function
265     Example 1 ::
266         ## 2D system
267         def derivs6(x,t):
268             d1 =  x[0] + 2*x[1]
269             d2 =  -3*x[0] + 4*x[1]
270             return (d1, d2)
271         dt = 0.0005
272         t = arange(0.0, 2.0, dt)
273         y0 = (1,2)
274         yout = rk4(derivs6, y0, t)
275     Example 2::
276         ## 1D system
277         alpha = 2
278         def derivs(x,t):
279             return -alpha*x + exp(-t)
280         y0 = 1
281         yout = rk4(derivs, y0, t)
282     If you have access to scipy, you should probably be using the
283     scipy.integrate tools rather than this function.
284     """
285
```

```
286     try:
287         Ny = len(y0)
288     except TypeError:
289         yout = np.zeros((len(t),), np.float_)
290     else:
291         yout = np.zeros((len(t), Ny), np.float_)
292 
293     yout[0] = y0
294 
295 
296     for i in np.arange(len(t) - 1):
297 
298         thist = t[i]
299         dt = t[i + 1] - thist
300         dt2 = dt / 2.0
301         y0 = yout[i]
302 
303         k1 = np.asarray(derivs(y0, thist, *args, **kwargs))
304         k2 = np.asarray(derivs(y0 + dt2 * k1, thist + dt2, *args, **kwargs))
305         k3 = np.asarray(derivs(y0 + dt2 * k2, thist + dt2, *args, **kwargs))
306         k4 = np.asarray(derivs(y0 + dt * k3, thist + dt, *args, **kwargs))
307         yout[i + 1] = y0 + dt / 6.0 * (k1 + 2 * k2 + 2 * k3 + k4)
308     return yout
```

Código-fonte 6.2: Implementação do AcroBot disponibilizada pela biblioteca do Gym

asad

REFERÊNCIAS

APPIAH, N.; VARE, S. **Playing FlappyBird with Deep Reinforcement Learning**. [S.l.]: Stanford University, 2018.

BAKER, M. **1,500 scientists lift the lid on reproducibility**. [S.l.]: nature, 2016. <https://www.nature.com/news/1-500-scientists-lift-the-lid-on-reproducibility-1.19970>. Accessed: 2019-11-20.

BARTO, A. G.; SUTTON, R. S.; ANDERSON, C. W. Neuronlike adaptive elements that can solve difficult learning control problems. **IEEE Transactions on Systems, Man, and Cybernetics**, SMC-13, n. 5, p. 834–846, Sep. 1983. ISSN 2168-2909.

BELLEMARE, M. G. et al. The arcade learning environment: An evaluation platform for general agents (extended abstract). In: **IJCAI**. [S.l.: s.n.], 2013.

BELLMAN, R. A markovian decision process. **Indiana Univ. Math. J.**, v. 6, p. 679–684, 1957. ISSN 0022-2518.

BERGES, V.-P.; CHEN, L. **Puppo, The Corgi: Cuteness Overload with the Unity ML-Agents Toolkit**. 2018. <https://blogs.unity3d.com/2018/10/02/puppo-the-corgi-cuteness-overload-with-the-unity-ml-agents-toolkit/>. Accessed: 2019-11-30.

Brockman, G. et al. OpenAI Gym. **arXiv e-prints**, p. arXiv:1606.01540, Jun 2016.

BURMEISTER, J.; WILES, J. The challenge of go as a domain for ai research: a comparison between go and chess. In: **Proceedings of Third Australian and New Zealand Conference on Intelligent Information Systems. ANZIIS-95**. [S.l.: s.n.], 1995. p. 181–186.

CASSANDRA, A. R. A survey of pomdp applications. In: **Thirteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI–97)**. [S.l.: s.n.], 1998. p. 472–480.

CATTO, E. **Box2D User Manual**. 2007. <http://box2d.org/manual.pdf>. Accessed: 2019-11-20.

CHEN, J. X. The evolution of computing: Alphago. **Computing in Science & Engineering**, v. 18, n. 4, p. 4–7, 2016. Disponível em: <https://aip.scitation.org/doi/abs/10.1109/MCSE.2016.74>.

CHEN, K. **Deep reinforcement learning for flappy bird**. [S.l.]: Stanford University, 2015.

COUMANS, E. **Bullet Real-Time Physics Simulation**. 2013. <http://bulletphysics.org>. Accessed: 2019-11-26.

GEBHARDT, G. H. et al. Learning to assemble objects with a robot swarm. In: **Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems**. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2017. (AAMAS '17), p. 1547–1549. Disponível em: <http://dl.acm.org/citation.cfm?id=3091282.3091357>.

GOODSTEIN, M.; ASHLEY-ROLLMAN, M.; ZAGIEBOYLO, P. **Parallelizing the Open Dynamics Engine**. 2007. <https://www.cs.cmu.edu/~mpa/ode/final_report.html>.

HEESS, N. et al. Emergence of locomotion behaviours in rich environments. **arXiv preprint arXiv:1707.02286**, 2017.

HESSE, C. **How to create new environments for Gym**. [S.l.]: GitHub, 2019. <https://github.com/openai/gym/blob/master/docs/creating-environments.md>.

HUTTER, M. et al. Anymal - a highly mobile and dynamic quadrupedal robot. In: **2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)**. [S.l.: s.n.], 2016. p. 38–44. ISSN 2153-0866.

JULIANI, A. et al. Unity: A general platform for intelligent agents. **arXiv preprint arXiv:1809.02627**, 2018.

KANG, D.; HWANGHO, J. **SimBenchmark**. [S.l.]: GitHub, 2018. <https://github.com/leggedrobotics/SimBenchmark>.

KAUFMAN, D. M. et al. Staggered projections for frictional contact in multibody systems. **ACM Trans. Graph.**, ACM, New York, NY, USA, v. 27, n. 5, p. 164:1–164:11, dez. 2008. ISSN 0730-0301. Disponível em: <http://doi.acm.org/10.1145/1409060.1409117>.

KEMPKA, M. et al. Vizdoom: A doom-based ai research platform for visual reinforcement learning. In: **2016 IEEE Conference on Computational Intelligence and Games (CIG)**. [S.l.: s.n.], 2016. p. 1–8.

KOENIG, N.; HOWARD, A. Design and use paragigm for gazebo, an open-source multi-robot simulator. In: IEEE. **2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No. 04CH37566)**. [S.l.], 2004. v. 3, p. 2149–2154.

LECUN, Y. et al. Object recognition with gradient-based learning. In: **Shape, contour and grouping in computer vision**. [S.l.]: Springer, 1999. p. 319–345.

MACHADO, M. C. et al. Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. **Journal of Artificial Intelligence Research**, v. 61, p. 523–562, 2018.

MINSKY, M. L. **Theory of neural-analog reinforcement systems and its application to the brain model problem**. Tese (Doutorado) — Princeton University, 1954.

MNIH, V. et al. Playing Atari with Deep Reinforcement Learning. **arXiv e-prints**, p. arXiv:1312.5602, Dec 2013.

NARASIMHAN, K.; KULKARNI, T.; BARZILAY, R. Language Understanding for Text-based Games Using Deep Reinforcement Learning. **arXiv e-prints**, p. arXiv:1506.08941, Jun 2015.

NG, A. Y. **Shaping and policy search in Reinforcement learning**. Tese (Doutorado) — University of Berkeley, 2003.

OPENAI. **FrozenLake-v0**. 2019. Disponível em: <https://gym.openai.com/envs/FrozenLake-v0/>.

PENG, X. B. et al. Deepmimic: Example-guided deep reinforcement learning of physics-based character skills. **ACM Trans. Graph.**, ACM, New York, NY, USA, v. 37, n. 4, p. 143:1–143:14, jul. 2018. ISSN 0730-0301. Disponível em: <http://doi.acm.org/10.1145/3197517.3201311>.

How to Beat Flappy Bird: A Mixed-Integer Model Predictive Control Approach, Volume 2: Mechatronics; Estimation and Identification; Uncertain Systems and Robustness; Path Planning and Motion Control; Tracking Control Systems; Multi-Agent and Networked Systems; Manufacturing; Intelligent Transportation and Vehicles; Sensors and Actuators; Diagnostics and Detection; Unmanned, Ground and Surface Robotics; Motion and Vibration Control Applications de **Dynamic Systems and Control Conference**, (Dynamic Systems and Control Conference, Volume 2: Mechatronics; Estimation and Identification; Uncertain Systems and Robustness; Path Planning and Motion Control; Tracking Control Systems; Multi-Agent and Networked Systems; Manufacturing; Intelligent Transportation and Vehicles; Sensors and Actuators; Diagnostics and Detection; Unmanned, Ground and Surface Robotics; Motion and Vibration Control Applications). V002T07A003. Disponível em: <https://doi.org/10.1115/DSCC2017-5285>.

SHEN, J. **Joints**. [S.l.]: GitHub, 2015. <https://github.com/GuidebeeGameEngine/Box2D/wiki/Joints>.

SILVER, D. et al. Mastering the game of go with deep neural networks and tree search. **Nature**, Nature Publishing Group, a division of Macmillan Publishers Limited. All Rights Reserved. SN -, v. 529, p. 484 EP –, Jan 2016. Article. Disponível em: <https://doi.org/10.1038/nature16961>.

Silver, D. et al. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. **arXiv e-prints**, p. arXiv:1712.01815, Dec 2017.

SIMS, K. Evolving virtual creatures. In: **Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques**. New York, NY, USA: ACM, 1994. (SIGGRAPH '94), p. 15–22. ISBN 0-89791-667-0. Disponível em: <http://doi.acm.org/10.1145/192161.192167>.

SMITH, R. **How to make new joints in ODE**. 2002. <http://www.ode.org/joints.pdf>. Accessed: 2019-11-20.

SMITH, R. **Open Dynamics Engine User Guide**. 2006. <http://www.gnu-darwin.org/www001/ports-1.5a-CURRENT/devel/ode-devel/work/ode-060223/ode/doc/ode.pdf>. Accessed: 2019-11-20.

SONDIK, E. J. **The Optimal Control of Partially Observable Markov Processes**. Tese (Doutorado) — Stanford University, 1971.

SUTTON, R. S. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In: **Advances in neural information processing systems**. [S.l.: s.n.], 1996. p. 1038–1044.

SUTTON, R. S.; BARTO, A. G. **Reinforcement learning: An introduction**. [S.l.]: MIT press, 2018.

TASSA, Y. et al. **DeepMind Control Suite**. [S.l.], 2018. abs/1504.04804. Disponível em: <https://arxiv.org/abs/1801.00690>.

THORNDIKE, E. L. Animal intelligence: An experimental study of the associative processes in animals. **The Psychological Review: Monograph Supplements**, v. 2, n. 4, p. i–109, 1898.

TODOROV, E.; EREZ, T.; TASSA, Y. Mujoco: A physics engine for model-based control. In: IEEE. **2012 IEEE/RSJ International Conference on Intelligent Robots and Systems**. [S.l.], 2012. p. 5026–5033.

TOGELIUS, J. et al. Super mario evolution. In: **2009 IEEE Symposium on Computational Intelligence and Games**. [S.l.: s.n.], 2009. p. 156–161.

TORREY, L.; SHAVLIK, J. Transfer learning. In: **Handbook of research on machine learning applications and trends: algorithms, methods, and techniques**. [S.l.]: IGI Global, 2010. p. 242–264.

WATTKINS, C. J. C. H. **Learning from Delayed Rewards**. Tese (Doutorado) — King's College, 1989.

ZENG, A. et al. TossingBot: Learning to Throw Arbitrary Objects with Residual Physics. **arXiv e-prints**, p. arXiv:1903.11239, Mar 2019.