Universidade Federal do ABC
Centro de Matemática, Computação e Cognição
Projeto de Graduação em Computação

Análise do uso de aprendizado por reforço e imitação para carros autônomos na Unity3D

Vítor Guilherme Antunes

Santo André - SP, Dezembro de 2023

Vítor Guilherme Antunes

Análise do uso de aprendizado por reforço e imitação para carros autônomos na Unity3D

Projeto de Graduação apresentada ao Centro de Matemática, Computação e Cognição, como parte dos requisitos necessários para a obtenção do Título de Bacharel em Ciência da Computação.

Universidade Federal do ABC – UFABC

Centro de Matemática, Computação e Cognição

Projeto de Graduação em Computação

Orientador: Fernando Teubl Ferreira

Santo André - SP

Dezembro de 2023

Agradecimentos

Agradeço primeiramente a minha família, por sempre me apoiarem e motivarem nos estudos.

Agradeço ao meu orientador, Fernando Teubl, por ter me acompanhado durante a realização deste trabalho.

Aos meus colegas e amigos que fiz durante a minha graduação na UFABC, que contribuiram bastante pro meu desempenho acadêmico e profissional.

Resumo

Veículos autônomos são um dos grandes objetivos da indústria automotiva atualmente. Embora muito progresso tenha sido feito, e testes nas ruas já sejam uma realidade, muitos desafios éticos estão no caminho para o aperfeiçoamento da tecnologia, por exemplo, como o veículo reagiria em caso de vida ou morte. Com isso em mente, este projeto se propõe de criar um simulador utilizando da Unity3D, uma game engine, para o treinamento de veículos autônomos, e que sirva de plataforma para estudos futuros podendo criar diversos cenários hipotéticos. Neste projeto, pretende fazer estudos comparativos de algoritmos de aprendizado por reforço e aprendizado por imitação no ambiente do simulador recém-criado.

Palavras-chave: veículos autônomos. Aprendizado por Reforço. Aprendizado por Imitação.

Abstract

Autonomous vehicles are one of the biggest goals of the automotive industry nowadays, although a lot of progress have been made and tests on road are already a thing, many ethical challenges are in the way of enhancement of this technology, such as how an autonomous vehicle would react in a life or death situation. With this in mind, this project aims to build a simulator using Unity3D game engine for the training of agents capable of driving a vehicle, which can be used as a platform for future studies around hypothetical scenarios. In this project, we intend to do early comparative studies about Reinforcement Learning and Imitation Learning algorithms in the newly developed simulation environment.

Keywords: Autonomous Vehicles. Reinforcement Learning. Imitation Learning.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

IA	Inteligência Artificial
AS	Aprendizado Supervisionado
RL	Reinforcement Learning (Aprendizado por Reforço)
IL	Imitation Learning (Aprendizado por Imitação)
MDP	Processo de Decisão de Markov
PGM	Policy Gradient Method (Método de polítca gradiente)
PPO	Proximal Policy Optimization
SAC	Soft Actor Critic
IRL	Inverse Reinforcement Learning
BC	Behavioural Cloning
GAIL	Generative Adversarial Imitation Learning

Lista de símbolos

$\in$	Pertence
$\doteq$	Se define por
S	Conjunto de estados
S^+	Conjunto de estados onde há estado terminal
A	Conjunto de ações
$A(s)$	Conjunto de ações válidas no estado s
π	Política
π_*	Política ótima
$v(s)$	função valor do estado s
$v_\pi(s)$	função valor do estado s sob política π
$q(s, a)$	função valor do ação-estado
$q_\pi(s, a)$	função valor do ação-estado sob política π
γ	Constante de desconto

Sumário

Introdução

No passado, veículos eram em grande parte máquinas mecânicas, com poucos recursos eletrônicos. Hoje em dia, diversos avanços foram feitos nos carros modernos, e estes já estão equipados com variadas tecnologias assistentes como controle de tração, freios ABS e de emergência, piloto automático, entre outros recursos que dependem de sensores e tomam decisões que controlam parcialmente o veículo, visando segurança e conforto ao condutor.

Veículos que possuem as assistências mencionadas acima comumente são chamados "autônomos", porém neste trabalho trataremos por "carro autônomo" um veículo capaz de conduzir-se sem depender de um humano. Conforme o padrão SAE J3016, a autonomia veicular é dividida em níveis que vão do zero a cinco, os recursos citados são no máximo nível 2, um veículo para ser "autônomo" seria de nível 3 ou maior (SAE (2014)), porém para atingir este grau de autonomia, é necessário mais do que sensores e controladores que dão instruções diretas ao carro.

Nas duas últimas décadas houve um aumento na presença de tecnologias como Inteligência Artificial e **Aprendizado de Máquina** no cotidiano das pessoas. Corretores ortográficos, reconhecimento facial, algoritmos de recomendação (leitura, música, compra, etc.) são alguns exemplos de diversos outros que até então não estavam disponíveis ao público, mas agora são amplamente aplicadas.

Existem três paradigmas de aprendizado de máquina: **supervisionado (AS)**, **não supervisionado** e **por reforço(RL)**. Este último aborda aprendizado que envolvam exercer uma atividade, nele, o **agente** durante seu treino, aprimora uma tarefa através de sucessivas tentativas e erros onde é premiado quando age corretamente e penalizado caso contrário. Este texto introduzirá os dois primeiros paradigmas e discorrerá mais detalhadamente sobre o **RL**. Embora o **AS** também seja utilizado autônomos, ele é mais usado em reconhecimento de elementos no ambiente, algo que está fora do escopo deste projeto.

Quando tenta-se criar um autômato que exerça uma atividade complexa como condução de um veículo, a abordagem clássica que usa algoritmos com condicionais e instruções diretas é insuficiente para tal, isto deve-se ao fato de que é inviável criar manualmente uma tabela de comandos dado certas condições, o tamanho da lista tenderia ao infinito. Utilizando de um paradigma de AM como **aprendizado por reforço** se faz necessário, pois o autômato dominará a técnica de conduzir um veículo após sucessivas tentativas.

Este projeto se propõe a criar um simulador que sirva de estudos para o aprimora-

mento de um agente condutor de carro autônomo. Isso compreende usar um modelo 3D de uma área urbana e de um carro e então realizar um estudo para desenvolver um condutor do veículo utilizando-se de RL.

Justificativa

Estes métodos de aprendizado de máquina requerem um "treinamento", onde o agente aprende sua tarefa através de tentativa e erro. Portanto, treinar um agente em vias públicas incorreria em riscos a terceiros. Logo, este trabalho visa criar um ambiente simulado, desta forma um agente poderia ser treinado em diversos cenários que simulem vias urbanas. Este ambiente também pode ser facilmente modificado para testar o aprendiz em diversas situações, o mesmo pode ter suas características alteradas sem muito custo. O agente poderia facilmente aprender em diferentes climas ou terrenos sem muito custo de deslocamento, ou tempo. Portanto, um simulador se mostra mais ideal para um estudo inicial do que um teste físico real mesmo que em ambiente fechado e fora das ruas públicas.

Objetivos

Objetivos Gerais

Neste projeto, será feito um estudo comparativo dos algoritmos de aprendizado por reforço e imitação. Com isso, o objetivo é obter uma perspectiva inicial dos algoritmos, além de validar se o simulador é adequado para estudos do ramo.

Objetivos específicos

Os algoritmos que o projeto visa comparar são: **Otimização de Política Proximal (PPO)**, **Clonagem Comportamental (BC)** e **Aprendizagem por Imitação Adversária Generativa (GAIL)**. Eles serão usados para treinar um agente de modo a faze-lo percorrer trajetos pré-definidos. O desafio exigirá que ele faça curvas, evite se chocar com obstáculos como subir em uma calçada ou colidir com uma parede. O agente que conseguir chegar ao destino sem cometer infrações em excesso é dito "eficiente" (critério do autor). Com isso o projeto visa responder as seguintes perguntas:

- Quais dos três algoritmos conseguiu gerar um agente que seja eficiente?
- Qual deles gerou o agente *mais* eficiente?
- Qual deles gerou este agente mais rápido?

Parte I

Preparação da pesquisa

1 Fundamentação teórica

Neste capítulo, serão introduzidos os conceitos e fundamentos das teorias e tecnologias utilizadas neste projeto. Será abordado sobre autonomia de veículos, como o que define um carro autônomo e os diferentes níveis de autonomia. Também abordará o que é um motor de jogo e a principal ferramenta que usaremos para desenvolver um ambiente simulado, a Unity 3D. Por fim, definição de IA, os paradigmas de aprendizado de máquina com um aprofundamento em aprendizado por reforço.

1.1 Veículos autônomos

A SAE (Society of Automotive Engineers) define 6 níveis de autonomia para veículos, do zero ao cinco (SAE (2014)). O primeiro, o nível zero, não possui automação de direção alguma, é limitado a apenas a tecnologias de assistência como freios ABS ou freio automático emergencial. Um veículo com automação de nível 1 já passa a possuir alguns recursos que ajudam na direção, como controle para manter o carro na via, controle de velocidade a fim de manter distância de outro veículo à frente.

Os níveis 2 e 3 são definidos por, respectivamente, “direção parcialmente automática” e “direção automática condicional”. A diferença de ambos pode ser bem sútil, pois em ambos os casos a automação teria o controle do volante, acelerador e freio. No primeiro seria apenas para atividades mais simples, como dirigir na estrada mantendo a velocidade e distâncias, enquanto no nível 3 o veículo assumiria o controle por um período, podendo até fazer manobras mais avançadas, como ultrapassar um automóvel mais lento à frente. Em ambos os níveis ainda é necessário a atenção máxima do condutor para assumir o controle quando for necessário.

Nos últimos dois níveis de autonomia, temos um veículo que seria capaz de exercer qualquer atividade sem intervenção humana, no nível 4 não seria necessário um motorista conduzindo, este só assumiria o controle quando quisesse, nesses níveis é capaz inclusive que o automóvel assuma o controle quando necessário, e o mesmo poderia ainda nem possuir volante ou pedais.

Atualmente, veículos com autonomia de nível zero a dois já estão presentes no mercado. Neste projeto estamos interessados em discutir sobre os carros autônomos de nível 3 ou mais, isto é, automóveis que possuam um alto nível de autonomia, o sistema deles podem ser divididos em quatro componentes, cada um deles usa aprendizado de máquina de formas diferentes, são eles: planejamento de rota, decisões comportamentais, controle de moção e controle veicular (Paden et al. (2016)).

O planejamento de rota se trata de definir o trajeto a ser percorrido, da origem ao destino. Representar a rede de ruas da cidade como se fosse um grafo ponderado e aplicar um algoritmo de busca como Djikstra não é prático. Então, traçar o melhor caminho até o destino requer algoritmos mais elaborados que levem em consideração dados como informação histórica e em tempo real.

Definida a rota, o veículo deve ser capaz de percorrer a mesma tendo em consideração todos os participantes do tráfego e as regras de trânsito, isso envolve uma complexidade de comportamento que vai além de apenas saber conduzir o veículo, então o componente de decisões comportamentais deve saber selecionar ações apropriadas para cada situação. A condução do automóvel em uma rodovia é diferente da condução em um cruzamento em uma rua urbana, onde ele deve parar o veículo e cruzar quando não há nenhum outro veículo ou pedestre no caminho.

A partir do momento que o comportamento foi selecionado, o controle de moção do automóvel autônomo que decide como este vai se locomover, mudança de faixa, conversão a direita, parar o veículo, avançar ao sinal verde, etc, todos essas ações devem ser realizadas de modo que não cause colisões, evitando obstáculos, e devem de ser realizada de forma que não cause desconforto aos passageiros.

O controle do veículo se define pelo controle dos atuadores mecânicos do automóvel para que tenha uma resposta em retorno, essa constante troca de informação é necessária para o controle de moção do veículo atue apropriadamente.

1.2 Motores de jogos

Um motor de jogo (do inglês *game engine*) é um *software* com o objetivo primário de se criar jogos eletrônicos. Estes programas incluem diversas ferramentas que auxiliam o desenvolvedor de jogos a criar seu produto, não existe uma definição sobre quais destas um software deve possuir para ser considerado um motor de jogo, mas frequentemente possuem módulos que lidam com *input* de usuário, renderização de gráficos 2D e 3D, som, motor de física (onde se lida com gravidade e colisão), animação, gerenciamento de memória entre outros (Andrade (2015)).

Como o objetivo é focado no desenvolvimento de uma inteligência artificial para um simulador de carro autônomo, um motor de jogo possui meios que facilitariam o trabalho. Não seria necessário desenvolver do zero a renderização dos modelos dos ambientes como veículos, ruas, calçadas, prédios, etc. Também não seria necessário desenvolver todo o complexo cálculo de física como força, gravidade, colisão de objetos, peso do veículo, entre outros. Isso permite que o foco do trabalho a ser feito se limite o máximo possível à análise da IA.

Inicialmente foi considerado o uso de outros motores de jogo, como a Unreal Engine 4 (software da Epic Games, Inc.) e a Godot Engine (código aberto sob licença MIT). Embora haja uma preferência natural por uma ferramenta de código aberto do que um software proprietário para realizar esta tarefa, a Unity 3D possui mais jogos publicados que a Godot Engine(ITCH.IO (2023)) e (SteamDB (2023)). Além disso, possui um acervo de recursos criados pela comunidade conhecido como *Unity Asset Store* onde é possível obter modelos 3D, efeitos sonoros e até modelos genéricos de jogos para se construir algo em cima disso.

A Unity 3D também possui uma biblioteca de ferramentas própria para criação de agentes inteligentes chamada de **Unity ML Agent Toolkit**, que foi escrita usando **Pytorch**, uma biblioteca **Python** para criação de redes neurais. A Unity ML Agent Toolkit oferece um aparato para se criar um agente inteligente, onde é configurado sensores e ações (o que seria a primeira e última camada de uma rede neural) podendo ser treinado usando aprendizado por reforço, por imitação, neuroevolução, entre outros.

1.2.1 Editor da Unity3D

Dentre as principais janelas da interface da Unity3D, temos: hierarquia, projeto, visualização da cena e inspetor, conforme mostra a figura 1. A primeira se trata dos elementos que há dentro da cena (pode-se entender como sendo um trecho do jogo, como este projeto contém apenas uma única cena não é necessário que o leitor entenda tudo que uma cena pode ser), estes elementos são chamados de *GameObjects*, eles são uma abstração de qualquer item dentro do jogo, incluindo o personagem que o jogador controla, o cenário com o qual interage, a "câmera" com o qual o jogo é visto também é um *GameObject*. A primeira janela é chamada de hierarquia porque o conjunto destes objetos podem formar uma estrutura em árvore, por exemplo, podemos criar um objeto "árvore" que conteria os objetos "raiz", "tronco" e "folhas" como seus objetos aninhados. Na seção da proposta é explicado a estrutura deste projeto com mais detalhes.

A janela de projeto estão os arquivos do projeto, é um diretório do projeto criado pela Unity3D onde deve conter arquivos de efeitos sonoros, códigos-fonte do desenvolvedor, fontes, modelos 2D e 3D, texturas, etc. A visualização da cena é uma janela que permite o desenvolvedor ter uma noção visual da disposição dos *GameObjects*, desta forma ele consegue posicioná-los mais apropriadamente, também consegue ver se a dimensão e escala deles estão coerentes.

A última janela é o inspetor, nela é exibido os detalhes dos *GameObjects*, isto é, os *components* destes. Como foi explicado anteriormente, os *GameObjects* podem assumir diversas funcionalidades, e são os componentes que permitem isso, por exemplo, o componente *Transform*, comum a todos os objetos da cena define as coordenadas (ou posição) que o objeto se encontrará na cena, sua rotação e escala, já os *components Mesh*

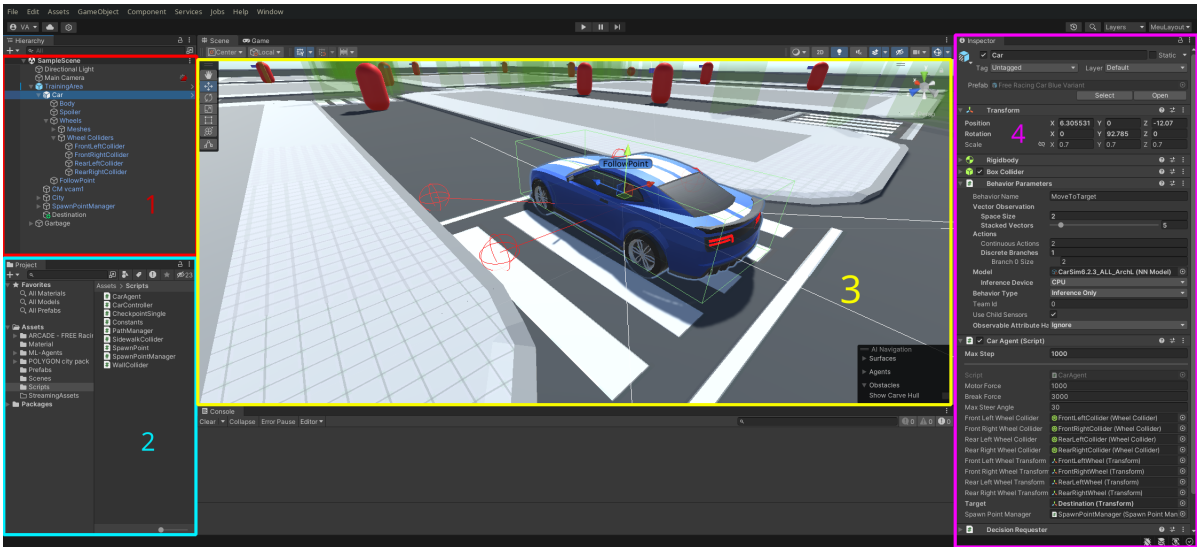

Figura 1 – Interface da Unity3D. A janela 1 é a hierarquia, 2 é a janela de projeto, 3 é visualização da cena e 4 é o inspetor

Filter e *Mesh Renderer* são responsáveis pela renderização 3D de um objeto, isto é, sua forma e aparência.

1.3 Inteligência Artificial

Não existe uma definição única sobre o que é Inteligência Artificial (IA), o mesmo pode ser dito quanto ao seu objetivo. De acordo com (Russell e Norvig (2009)), pode-se dizer que se resume em criar um programa de computador que faça uma ou mais dos seguintes tópicos: pensar humanamente, agir humanamente, pensar racionalmente e agir racionalmente. Pensar pode ser entendido como o processo de pensamento, de compreender e argumentar enquanto agir pode ser entendido como tomar ações e possuir um certo comportamento. Humanamente mediria o quanto a Inteligência Artificial consegue se aproximar de um desempenho humano (agindo ou pensando), e racionalmente é quando há o interesse em pensar ou agir de forma ideal, isto é, que faça "a coisa certa".

Uma IA que age racionalmente pode ser entendido como um programa de computador que toma decisões autonomamente. De fato, qualquer algoritmo pode ser criado para tomar decisões de acordo com uma série de condições, mas é esperado mais de um **agente racional**, ele é criado para observar o ambiente em que está inserido e tomar decisões por um período prolongado, adaptando-se a qualquer mudança e procurando cumprir seu **objetivo** fazendo isso de forma ideal, não cometendo erros, e quando estes forem inevitáveis, deverá agir de forma a minimizar o dano causado (Russell e Norvig (2009)).

Para desenvolvimento de um veículo autônomo que este trabalho se propõe a estudar, não basta um algoritmo definindo condições e instruções, isto seria uma abordagem insuficiente tanto em eficiência quanto em praticidade em seu desenvolvimento. Seria então

necessário um **agente racional**, que possua as características descritas no parágrafo anterior, que saiba observar o ambiente em sua volta e *aprenda* a agir de acordo. A seção abaixo elabora mais sobre esta área específica da Inteligência Artificial.

1.3.1 Aprendizado de máquina

Aprendizado de máquina é qualquer processo automatizado que tem como objetivo reconhecer um padrão dentro de um conjunto de dados (Kelleher, Namee e D'Arcy (2015)), em aprendizado de máquina o projetista cria um agente-aprendiz, fornece os dados e define um objetivo a fim de fazer seu agente melhorar seu desempenho na tarefa após sucessivas iterações observando os dados o tomando decisões.

Há dois paradigmas em AM que valem a pena ser mencionados brevemente antes de irmos ao que será utilizado no projeto, são eles: Aprendizado supervisionado, não-supervisionado. O primeiro lida mais com problemas de classificação, ao agente é fornecido dados rotulados e, ao analisá-los, se o treino foi efetivo, o agente seria capaz de atribuir um rótulo a um novo registro com uma alta acurácia. O Aprendizado não-supervisionado envolve dados sem rótulos, o objetivo do agente é encontrar uma estrutura oculta que conecte os dados, este processo é chamado de clusterização. Embora possa haver espaço para estes paradigmas no campo de autonomia veicular, eles não estão no escopo deste projeto.

1.4 Aprendizado por reforço

No aprendizado por reforço o objetivo é sempre fazer com que o agente aprenda a executar uma tarefa, se trata de ensiná-lo a como fazer. O agente está inserido em um ambiente e a ele é dado um objetivo, com isso ele deve aprender a tomar as decisões corretas nas situações apropriadas visando seu propósito. O aprendiz não é instruído sobre quais ações tomar em dadas circunstâncias, ao invés disso o agente aprenderá a tomar as decisões com base na orientação do treinamento, que envolve em premiá-lo quando agir idealmente ou penalizá-lo caso contrário. Esse parecer que o agente recebe é um valor numérico a ser maximizado por ele, e o dever do projetista é programar os critérios que decidem não somente o que é recompensador ou penalizador, mas o quanto é.

A seguir será formalizado como esse proceso ocorre, toda esta seção é baseada em Sutton e Barto (2018), exceto quando abordarmos sobre os algoritmos de otimização de política. É importante esclarecer aqui que a formalização a seguir se trata de tipos de problemas mais simples do que iremos lidar neste projeto, porém essa introdução é necessária para compreender os conceitos mais avançados serão vistos posteriormente.

1.4.1 Processo de tomada de decisão

Neste paradigma, o *agente* toma uma decisão A_t que afeta o *ambiente* o qual está inserido, que retorna a ele um estado S_{t+1} e uma recompensa R_{t+1}. O conjunto S representa todos os estados possíveis do ambiente, cada estado s representa toda a observação que o agente tem do sistema.

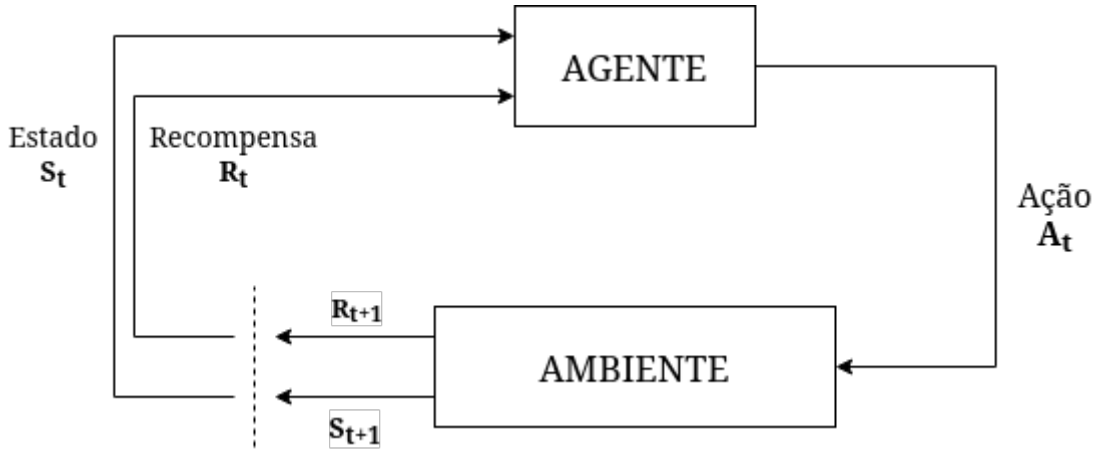

Figura 2 – diagrama da interação do agente com o ambiente. Adaptado de Sutton e Barto (2018)

Usando o jogo da velha de exemplo, o conjunto S representaria todas as configurações possíveis do tabuleiro, um estado $S_t \in S$ seria um configuração específica, sendo que S_0 poderia representar o tabuleiro vazio antes de qualquer jogada. Uma ação $a \in A$ seria uma jogada e por fim $t \in \{0, 1, 2, 3, ..., T\}$ as etapas do jogo até a etapa final T.

Essa sequência de estados, ações e recompensas é chamada de *trajetória* e pode ser escrita da seguinte forma:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, ..., R_n, S_n \tag{1.1}$$

Esse sistema é conhecido como **Processo de Decisão de Markov** (**MDP**, do inglês Markov Decision Process). É uma formalização de problemas que possuem um número *finito* de estados, ações e recompensas, podendo ser **estocástico**, isto é, não é possível antecipar com exatidão qual será S_{t+1} dado S_t e A_t e de **tempo discreto**. Dito isso, temos a seguinte distribuição de probabilidade p de um estado s' ocorrer após o agente tomar a decisão a no estado s:

$$p(s', r|s, a) \doteq Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\} \tag{1.2}$$

E por ser uma distribuição de probabilidades, temos:

$$\sum_{s' \in S} \sum_{r \in R} p(s', r|s, a) = 1, \text{ para todo } s \in S, a \in A(s) \tag{1.3}$$

Sendo $A(s)$ é o conjunto de todas as ações que podem ser tomadas no estado s.

Em um MDP, a probabilidade de cada valor de S_t e R_t depende apenas de S_{t-1} e A_{t-1} e de nenhum outro estado anterior a este. Essa caraterística é chamada de **propriedade de Markov**.

O comportamento do agente é moldado pelas recompensas que recebe a cada ação que toma, este sinal de recompensa é o que indica se o agente está executando a tarefa apropriadamente ou não. Diferente dos estados S e ações A, que podem assumir qualquer forma, o conjunto R é um conjunto de números reais: $r \in R \in \mathbb{R}$. A fórmula que deve ser maximizada é chamada de **retorno esperado**, é denotado por G_t:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T, \tag{1.4}$$

onde T é a última etapa que ocorre quando se atinge um **estado terminal**. Nem todas as tarefas possuem um estado terminal, mas as que têm são chamadas de **tarefas episódicas**. Usando o exemplo do jogo da velha, podemos entender que cada estado que há um vencedor ou que não há mais jogadas possíveis, seja um estado terminal. Em contraste, se uma tarefa não possui fim, é chamada de **tarefa contínua**. Como tarefas contínuas teriam $T = \infty$ logo $G_t = \infty$ e se torna impossível calcular por um computador. Então para isso é aplicado um desconto γ na equação onde $0 \leqslant \gamma \leqslant 1$:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{1.5}$$

A **constante de desconto** γ representa o valor de recompensas futuras, caso $\gamma = 0$ o agente apenas se preocuparia em maximizar R_t independente dos valores de recompensas futuras. Quando se aplica um γ se aproxima de 1, mais se valoriza estados futuros e aumenta as chances de o agente tomar uma ação que tenha uma recompensa imediata menor mas maiores recompensas futuras. Esta constante não é aplicada somente em problemas contínuos, ela também é útil quando o número de estados é muito grande e é necessário que o agente considere estados futuros.

Com isso chegamos a uma das partes centrais do AR, que é **funções valores**. Uma função valor é o que determina o quão importante é para o agente estar naquele estado, isto é, seu valor que é determinado pelo retorno esperado de recompensas futuras daquele estado. Para poder calcular isso é preciso saber como o agente opera, logo, é preciso saber sua **política**.

A **política** (representada por π) é o mapeamento do estado para a ação a ser tomada, formalmente, $\pi(a|s)$ é a probabilidade de se tomar a ação a no estado s. Aprendizado por reforço é, basicamente, aprimorar a política do agente e a mesma tende a mudar de uma trajetória para outra (caso episódico) ou até mesmo dentro de uma trajetória.

A definição de uma função v valor do estado s sob uma política π é:

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s] \tag{1.6}$$

Da mesma forma, pode-se estimar o valor de uma ação, para isso há a **função valor da ação** $q_\pi(s, a)$, definida da seguinte forma:

$$q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \tag{1.7}$$

As funções valores v_π e q_π podem ser estimadas através de experiência. Pode-se começar com uma política que basicamente atribui a mesma probabilidade para qualquer ação em qualquer estado, conforme as recompensas vão sendo aplicadas o valor esperado do estado e da ação serão atualizados.

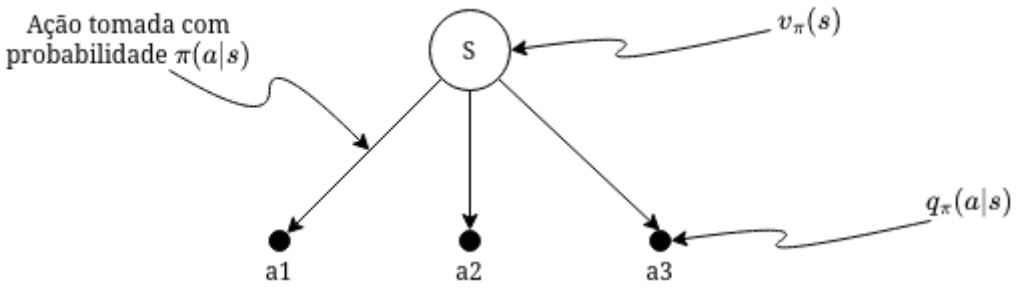

Figura 3 – Diagrama das funções valores. Ao estado s é atribuído o valor $v_\pi(s)$, a cada uma das ações é atribuído o valor $q_\pi(a_n|s)$ e tem probabilidade $\pi(a_n|s)$ de ser selecionada. Adaptado de Sutton e Barto (2018)

Sabemos que G_t é uma função recursiva de modo que $G_t \doteq R_{t+1} + \gamma G_{t+1}$. Com isso é fácil percerber que o mesmo se aplica para as funções valores, para qualquer política π em qualquer estado s, temos a seguinte função valor recursiva:

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma \mathbb{E}[G_{t+1}|S_{t+1} = s']] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')], \text{ para todo } s \in S \end{aligned} \tag{1.8}$$

Resolver uma tarefa por aprendizado por reforço é, grosseiramente, encontrar uma política que retorne uma grande quantidade de recompensa no longo prazo. Um política π é dita melhor que outra π' se e somente se $v_\pi(s) \geqslant v_{\pi'}(s)$, para todo $s \in S$. Em um MDP finito, é garantido que haja pelo menos uma **política ótima**, isto é, uma que seja melhor que qualquer outra para todos os estados, esta política é geralmente representada por π_*, neste caso ela deve utilizar uma *função valor do estado* também ótima:

$$v_*(s) \doteq \max_\pi v_\pi(s), \text{ para todo } s \in S \tag{1.9}$$

similarmente, para acharmos a *função valor da ação* ótima:

$$q_*(s,a) \doteq \max_\pi q_\pi(s,a), \text{ para todo } s \in S \text{ e } a \in A(s) \tag{1.10}$$

1.4.2 Algoritmos de otimização de política e conceito de aproximação

Nesta seção será abordado mais sobre os diferentes tipos de algoritmos de otimização de política, principalmente o PPO, SAC que estão disponíveis no ML-Agents da Unity 3D.

Existem muitas formas de se chegar aos valores das funções (1.9) e (1.10) e na política ótima, dentre elas pode-se citar programação dinâmica usando a equação (1.8) que é uma **equação de Bellman**, onde usa-se iteração para avaliar os estados e depois otimizar a política, e como a política otimizada reavalia os estados para então otimizar a política novamente, e assim por diante até que os valores dos estados convirjam para seus "valores verdadeiros".

Outro método seria **Monte Carlo**, que não assume total conhecimento do ambiente (condição necessária para otimizar com programação dinâmica) e utiliza-se de experiência prévia. Por fim temos **Q-learning** que combina um pouco dos dois. Apesar destes métodos terem suas aplicações, não discorreremos aqui sobre eles, pois, como foi explicado anteriormente, a tarefa que o agente condutor executará não é um MDP finito.

A condução de um veículo autônomo não possui um número finito de estados, pelo contrário, se considerarmos o uso de visão computacional com processamento de imagem a quantidade de estados possíveis se torna virtualmente infinito. Os problemas que um MDP finito resolve são **tabulares**, isto é, podem ser dispostos em uma tabela, onde se guardaria a relação com ações válidas, seus valores, etc.

Para isso há os **métodos de aproximação**, onde a quantidade de estados possíveis é tão grande que podemos assumir que o agente nunca passará pelo mesmo estado novamente. Portanto, uma abordagem em que alterar um valor $v(s)$ não altera o valor de outro estado semelhante $v(s')$ não parece útil, ao invés disso precisamos assumir que a valorização de um estado implique na valorização de outro. Além disso, temos o fato de que é inviável calcular $v_*(s)$ para todo $s \in S$, então agora será explorado os métodos de **aproximação**, que se baseia em utilizar uma função valor do tipo $\hat{v}(s, \mathbf{w}) \approx v_*(s)$ e $\mathbf{w} \in \mathbb{R}^d$. Dessa forma $\hat{v}$ pode ser tanto uma função linear com $\mathbf{w}$ sendo um vetor dos coeficientes quanto uma rede neural com $\mathbf{w}$ sendo os pesos das conexões.

1.4.3 Métodos de gradiente de política e de ator-crítico

Métodos de gradiente de política são uma abordagem diferente das já mencionadas acima, no caso a atualização da política não depende das funções valores das ações, em vez disso temos $\pi(a|s,\theta) = \Pr\{A_t = a|S_t = s, \theta_t = \theta\}$ para a probabilidade de ação a ser selecionada no tempo t dado que o ambiente se encontra no estado s com parâmetros $\theta \in \mathbb{R}^{d'}$. Agora o interesse é atualizar θ a fim de aprimorar a política da seguinte forma:

$$\theta_{t+1} = \theta + \alpha\widehat{\nabla J(\theta_\mathbf{t})}, \tag{1.11}$$

onde α é o tamanho do salto do gradiente e $\widehat{\nabla J(\theta_\mathbf{t})} \in \mathbb{R}^d$ é uma estimativa estocástica cuja expectativa se aproxima do gradiente da medida de desempenho em relação ao seu argumento θ_t. Qualquer método que obedeça a regra acima é um **método de gradiente de política** (PGM, do inglês Policy Gradient Method) podendo ou não utilizar de aproximação de função valor do estado $\hat{v}(s)$ descrito acima, caso utilize será considerado um **método de ator-critico** (actor-critic method).

Como foi mencionado na introdução, o pacote ML-Agents possui dois algoritmos de aprendizado por reforço: o **PPO** e o **SAC**. O primeiro é a sigla para **Proximal Policy Optimization** e é um PGM que tem o diferencial de que ele alterna entre extrair dados da política e executar uma otimização em lotes dos dados extraídos, além de aproveitar de se aproveitar de TRPO (Trust Regional Policy Optimization) mas sem ser tão complicado quanto. O algoritmo foi criado a fim de ser mais eficiente em processamento de dados e mais robusto (obter sucesso sem ter que ajustar os hiper-parâmetros) (Schulman et al. (2017)).

SAC (ou Soft Actor-Critic) é um algoritmo to tipo off-policy que visa maximizar o retorno esperado e entropia, isto é, obtem progresso agindo mais aleatoriamente possível. Isso é para evitar um problema comum em métodos que lidam com algoritmos de aprendizado de reforço que são livres de modelo: fragilidade a hiper-parametros (quando uma pequena mudança nestes gera um distúrbio muito grande no desempenho do treino)(Haarnoja et al. (2018))

1.5 Aprendizado por Imitação

Um dos problemas que ocorre com Aprendizado por Reforço é a sua ineficiência no começo do aprendizado. Devido a suas ações serem randômicas, existe uma enorme dificuldade em fazer um agente aprender uma tarefa executada por humanos. Para isso há o Aprendizado por Imitação (**IL**, do inglês Imitation Learning) que são métodos os quais o agente adquire habilidades ou comportamentos ao observar uma demonstração da dada

tarefa executada por um especialista. Neste paradigma o agente consegue moldar uma política geral para baseado no conjunto de dados vindos do demonstrador.

Uma abordagem de **IL** é a **Clonagem de comportamental** (**BC**, do inglês *Behavioural Cloning*), que utiliza-se de aprendizado supervisionado, onde os dados são os estados e os rótulos seriam as ações. O conjunto de dados fornecidos pelo especialista seria então um conjunto de trajetórias D composto por pares (S_t, a_t), onde S_t é um vetor de observações do agente e o a_t seria a ação tomada (rótulo)(Hussein et al. (2017)).

O problema com Aprendizado por Imitação é que sozinho não é suficiente para resolver problemas mais complexos, especialmente se envolvem ações contínuas. Erros na demonstração e uma generalização ruim podem fazer com que o agente não encontre uma política ótima ficando preso em um mínimo local. Quando a complexidade do problema aumenta, com vetores de muitas observações contínuas, encontrar durante o treino o mesmo ou um estado semelhante a um do conjunto de demonstração se torna improvável, levando a uma incapacidade de generalizar a política e consequentemente a um desempenho fraco (Ho e Ermon (2016)).

Um conceito dentro de IL é Aprendizado por Reforço Inverso (**IRL**, do inglês *Inverse Reinforcement Learning*), seu propósito é extrair a função de recompensa dado um comportamento (ótimo) observado nas demonstrações. IRL é ideal quando não é possível ou é difícil criar uma função de recompensa manualmente(Russell (1998)), então com as demonstrações o algoritmo busca uma aproximação linear a seguinte função:

$$R(s) = \alpha_1\phi_1(s) + \alpha_2\phi_2(s) + ... + \alpha_d\phi_d(s), \tag{1.12}$$

sendo $\phi_1, ..., \phi_d$ são funções do tipo $S \mapsto \mathbb{R}$ e os α_i's são os parâmetros que queremos ajustar. Aproximando esta função de recompensa, o IRL consegue encontrar a política ótima do especialista.

Com isso podemos apresentar o último algoritmo que será utilizado neste projeto: **GAIL** (do inglês *Generative Adversarial Imitation Learning*) (Ho e Ermon (2016)). Este algoritmo utiliza-se de GANs (*Generative Adversarial Networks*) removendo assim a restrição de que o custo pertence a uma classe limitada de funções, ao invés disso permite que seja aprendido usando aproximadores de funções expressivas como redes neurais. Além disso utilizando-se de TRPO (*Trust Region Policy Optimization*), GAIL consegue encontrar diretamente a política em vez de funções valores intermediárias (Bhattacharyya et al. (2020)).

2 Estado da Arte

Neste capítulo, serão discutidos e analisados trabalhos que envolvem a criação de agentes inteligentes em ambientes virtuais, a primeira seção será dedicada àqueles que não utilizam o pacote ML-Agents. Na outra seção, porém será visto projetos que se utilizam da biblioteca da Unity3D.

TORCS, CARLA e ALVINN

Um dos primeiros e mais conhecidos projetos que envolve direção autônoma é o ALVINN (Pomerleau (1991)), nele foi utilizado aprendizado por imitação, onde usaram de uma câmera posicionada no painel do veículo e um humano dirigindo, os dados gerados do especialista foram usados para treinar a rede neural. O veículo conseguiu percorrer uma distância de 400 metros a uma velocidade de 1,5 m/s.

TORCS é um simulador de código aberto bastante modular que pode ser usado para a criação de agentes artificialmente inteligentes. Sua portabilidade e modularidade é ideal para poder modificar o ambiente de modo a aumentar os desafios dos agentes que serão treinados nele, porém seu escopo se limita a pistas de corridas, então não é o ideal para percorrer trajetos urbanos que contenha pedestres, semáforos, regras de trânsito, etc (Wymann et al. (2013)).

Um projeto muito similar ao que este trabalho se propõe a fazer é o **CARLA**. Car Learning to Act (CARLA) é um ambiente de código aberto de treinamento condução autônoma de veículos terrestres, foi desenvolvido usando a **Unreal Engine 4**. A simulação inclui clima dinâmico, uso de visão computacional, obstáculos dinâmicos como carros e pedestres além de visual fotorrealista. Quanto ao treinamento utiliza-se de três métodos: um fluxo modular, aprendizado por imitação e aprendizado por reforço, treinaram os agentes em cada método em 4 níveis diferentes de dificuldade, e então postos a testes em climas e uma cidade distinta. O resultado foi surpreendente, com o aprendizado por reforço tendo um desempenho muito inferior aos outros dois métodos (Dosovitskiy et al. (2017)).

Criação de agentes inteligentes utilizando ML-Agents

O projeto desenvolvido por (Urrea, Garrido e Kern (2021)) também envolve um carro autônomo, apesar de não envolver um ambiente urbano e sim uma pista em formato de 8 (para treino) e outra com começo e fim (para teste). O trabalho analisa os dados da dirigibilidade comparando um humano em um simulador, um agente criado usando

aprendizado por imitação e outro agente utilizando **RL**.

Outro trabalho foi feito por (MÉXAS (2021)), que também se trata de um veículo autônomo, mas desta vez é um veleiro. Neste projeto, o autor faz um estudo comparativo dos algoritmos **PPO** e **SAC** para aprendizado por reforço, e os algoritmos **BC** e **GAIL** para **IL**. O ambiente dele considera a ação do vento sobre o veleiro e ao analisar os dados conclui que o primeiro algoritmo de **RL** é o que apresentou o melhor desempenho.

Utilização de Aprendizado por Reforço e Imitação em jogos

Outro projeto desenvolveu e analisou modelos de para controle de jogos eletrônicos utilizando-se de RL e IL (Souza (2023)). O jogo utilizado para a análise foi o *Sonic: The Hedgehog*, gerando modelos com GAIL e BC com PPO para otimização de política, os mesmos algoritmos que será utilizado neste projeto. O jogo possui diversos níveis nos quais os modelos foram treinados, que variam em extensão e complexidade, alguns deles requerem que pequenos *puzzles* sejam solucionados para serem concluídos. No fim dos testes, os modelos que se utilizaram de IL obtiveram um desempenho melhor, além de um treino com convergência mais rápida.

Parte II

Metodologia

3 Ferramentas

Para criar o projeto foi usado a Unity3D na versão **2022.3.7f1**. Acerca do **ML Agents**, há dois pacotes, o primeiro para o editor da Unity3D que adiciona os componentes e classes de **RL** e **IL**, esta está na versão **2.0.1**. O outro pacote seria o pacote Python **mlagents-learn**, instalado via **pip** (gerenciador de pacotes do Python), este encontra-se na versão **0.30.0**.

Como foi dito, para fazer o agente treinar é necessário criar um ambiente Python que interage com o editor da Unity informação, para isto foi usado Python na versão **3.9.13**, a principal dependência Python do **mlagents** é o *framework* **PyTorch**, neste projeto usa-se a versão **1.7.1**. As demais dependências Python estão presentes no requirements.txt disponíveis online no repositório oficial deste projeto que se encontra em https://github.com/antunesvitor/SimuladorDeConducao

A Unity e o Python estão presentes em para Windows e Linux, ambos os sistemas operacionais foram usados para o desenvolvimento deste projeto, porém grande parte dos testes foram conduzidos no **Arch Linux** utilizando-se do Kernel **6.4.10-arch1-1**.

No repositório mencinado acima encontra-se também instruções de instalação tanto no **Windows** quanto no **Arch Linux**.

Para a criação do ambiente, foi utilizado dois *assets* da Unity Asset Store, ambos fornecem os modelos 3D necessários para montar o ambiente de treino. O primeiro fornece modelos de carros que servirão como o agente (Unity Asset Store (2023a)) o outro contém diversos modelos necessários para a construção da cidade como ruas, calçadas, prédios, residências, árvores, postes, etc. (Unity Asset Store (2023b))

4 Modelo proposto

Este capítulo abordará o sistema de treinamento proposto. Primeiramente será exposto como funciona todo o ambiente desenvolvido, depois será explicado sobre a configuração dos algoritmos; por fim é explicado como será o treinamento e teste dos agentes.

4.1 Modelo

Aqui é descrito sobre o modelo do projeto. Por "modelo" entende-se o conjunto ambiente-agente-recompensas. Cada um deles é detalhado nas subseções abaixo. Dentre os módulos que compõe um veículo autônomo o foco deste projeto será em desenvolver o componente de controle de deslocamento que é responsável por fazer carro percorrer um trajeto. No mundo real as rotas seriam geradas por outro componente, porém isto está além do escopo deste projeto, portanto elas serão definidas pelo autor. Importante ressaltar que o conjunto de rotas deve ser composto por uma variedade de trajetos que exijam uma diversidade de manobras a serem aprendidas pelo agente.

4.1.1 O ambiente

A simulação envolve treinar o veículo para percorrer trajetos em ambiente urbano, a princípio, por uma questão de simplicidade, o agente não terá de lidar com declives ou aclives, semáforos, outros veículos ou pedestres. Portanto, o foco poderá se manter no estudo de fazer um agente percorrer as rotas. Nas bordas do cenário há muros que limitam o alcance do veículo. Como os trajetos desta proposta são curtos, não há necessidade de um modelo muito grande, porém pode vir a ser expandido em estudos futuros que visem trajetos longos. O agente recebe uma punição ao se chocar com a calçada ou com os muros da borda da cidade. Na Figura 5, é possível ver os muros ao fundo e as calçadas à direita; e a Figura 4 exibe uma visão superior de todo o cenário.

As rotas são compostas por: origem, *checkpoints* e destino. O primeiro indica o local de partida do veículo, o último é o objetivo final do agente naquele episódio. Os *checkpoints* são barreiras que indicam o caminho que deve ser percorrido, cada vez que o veículo atravessa uma delas ele ganha uma recompensa, isto é uma forma de indicar a ele que está fazendo o correto. Na Figura 5, há os *checkpoints* identificados pelas barreiras verdes semitransparentes. Há 17 percursos predefinidos, cada um deles possuem características distintas como distância origem-destino, quais e quantas conversões a serem feitas, isto foi moldado visando trazer uma diversidade maior de desafios a serem superados pelo agente.

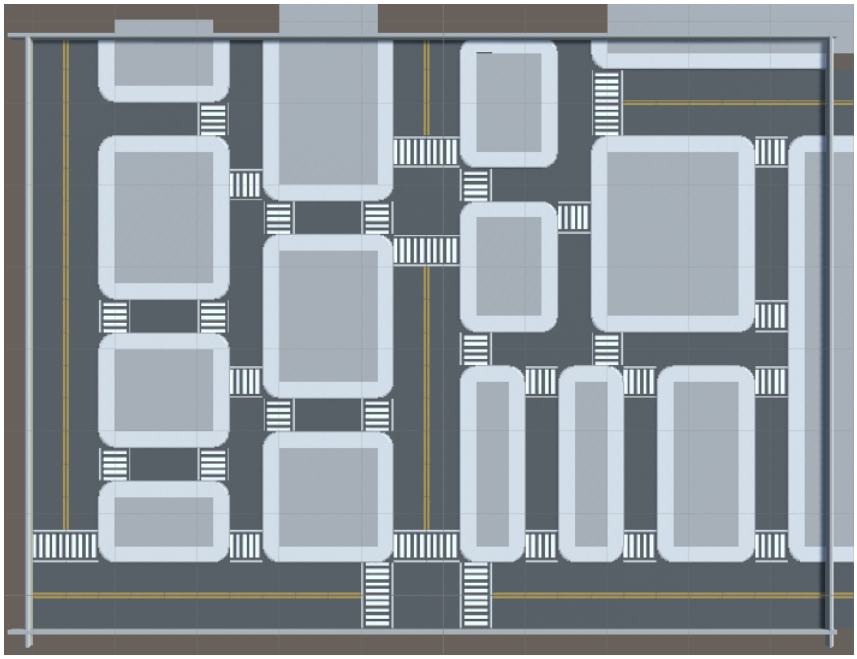

Figura 4 – Visão superior o cenário urbano criado para o treinamento do veículo

Assim como o tamanho do cenário, novos trajetos podem ser considerados em estudos futuros.

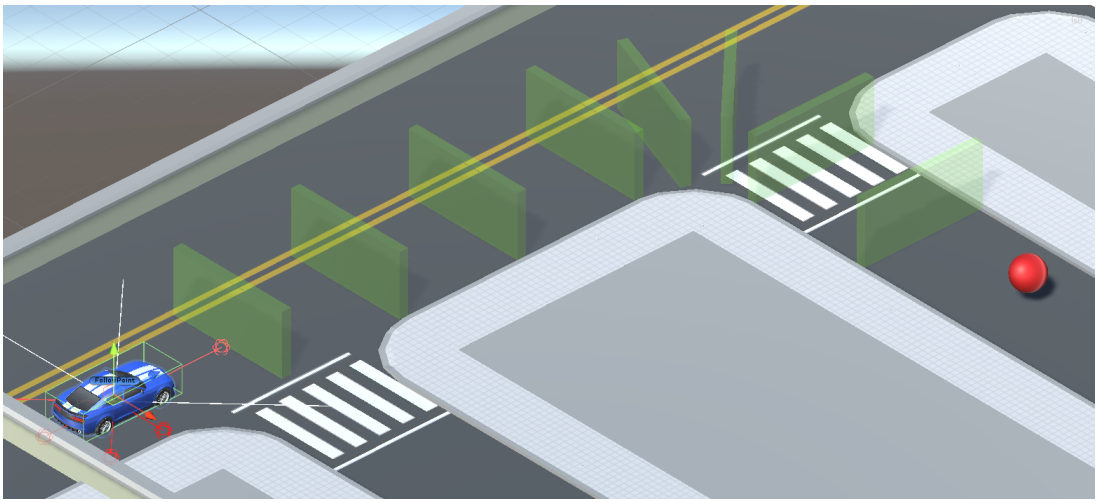

Figura 5 – Rota vista de perspectiva isométrica, o agente posicionado a origem ao canto esquerdo, com os *checkpoints* ao longo do percurso até o destino no canto direito.

4.1.2 O agente

O agente possui 20 sensores de distância divididos em 2 conjuntos. Estes sensores são um *component* do pacote **ML-agents** da Unity3D e além da distância são capazes de distinguir qual o objeto. Estes sensores dentro do editor são representados por feixes que partem do centro do veículo, é possível configurar diversos atributos deles como a quantidade, o ângulo máximo de distância do primeiro ao último, o ângulo vertical (que

indica se eles apontam para cima ou para baixo) e o tamanho da esfera, que nada mais é que a tolerância de colisão do sensor.

O primeiro conjunto fica à altura da visão do motorista e servem principalmente para detectar os *checkpoints*. Este possui 9 sensores, um frontal e 8 dispostos em uma faixa de 80° (4 para cada lado a partir do frontal). O segundo conjunto fica mais próximo ao solo, seu propósito principal é detectar a calçada e as paredes. Este possui 11 sensores, um frontal e 10 dispostos em uma faixa de 170°, assim ele é possível detectar obstáculos nas laterais do veículo bem como na parte traseira. Na figura 6, é possível ver os feixes partindo do carro, em verde são os que pertencem ao primeiro conjunto se chocando em um *checkpoint*. Em vermelho, está o segundo conjunto com alguns de seus raios se chocando a parede.

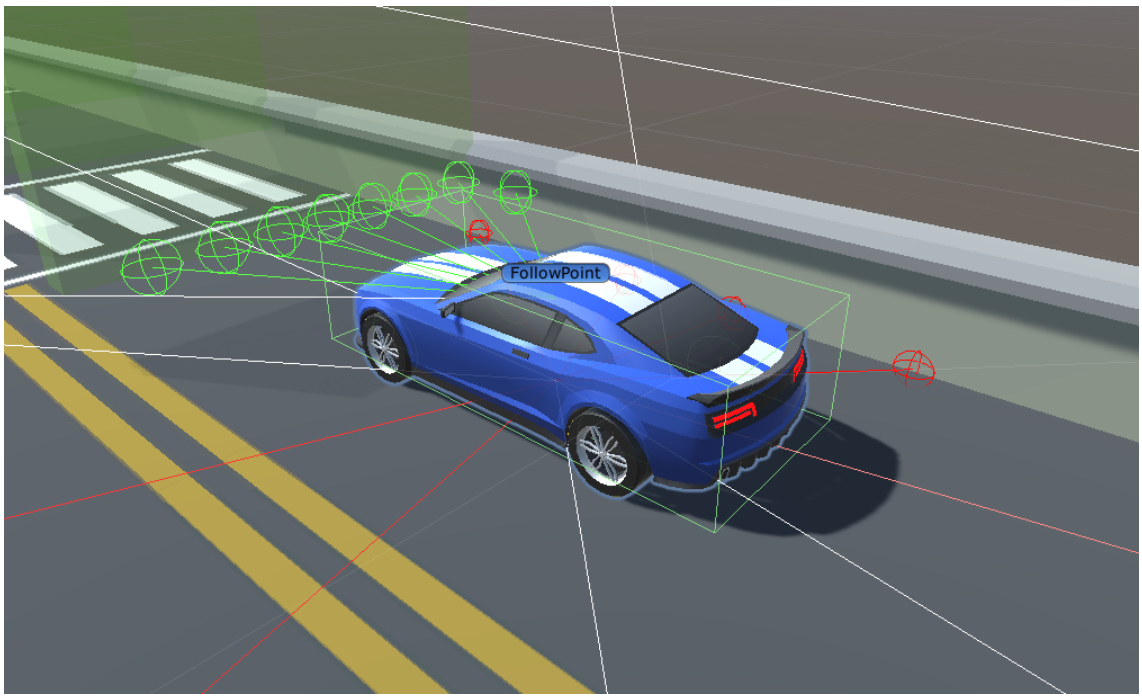

Figura 6 – Exemplo do agente e seus raios perceptores, é possível ver os dois conjuntos de sensores se chocando com objetos do ambiente.

A quantidade de sensores e sua distribuição foram decididas após alguns testes manuais do autor, com a intenção de evitar pontos cegos porém sem um número excessivo de observações causando lentidão no treinamento. Além dos raios perceptores, foi adicionado a velocidade do veículo à lista de observações do agente. Vale lembrar que o não foi implementado um recurso de imagem em um primeiro momento, ou seja, o veículo só enxerga através dos sensores e tem ciência de sua própria velocidade, ele é "cego" a qualquer outro elemento do ambiente que não esteja se choccando com o sensor.

Quanto as ações que o agente pode fazer são apenas 3: acelerar, frear e direcionar as rodas a direita e a esquerda. A primeira e a última ação são chamadas ações contínuas, sendo assim um número que indica o nível de aceleração que o carro irá produzir, e no

caso da direção da roda o quão direciona elas estão a esquerda ou direita. O ato de frear por outro lado é uma grandeza discreta, o veículo está ou não está freando.

4.1.3 As recompensas

Para compreender as recompensas é necessário distinguir as recompensas que são aplicadas durante o episódio das que são aplicadas ao fim do mesmo. Ambas são importantes para a otimização do comportamento do agente. A política do agente é atualizada dentro do episódio e também um após o outro, por isso receber a recompensa ou punição durante o episódio é necessário para que o agente saiba quais estados devem ser almejados e quais devem ser evitados.

Os eventos sujeitos a aplicação de recompensa ou punição que não definem o fim do episódio e sua representação matemática são os seguintes:

- Passar por um *checkpoint* (r_c);
- Colidir com a parede (p_p);
- Colidir com a calçada (p_c);
- *Step* (p_{step})

Os *steps* são as etapas explicadas no capítulo anterior, cada um possui um estado em que o agente se encontra e permite que ele tome uma decisão. O p_{step} é importante para que o agente não fique parado, como se estivesse "receoso" de receber uma punição, aplicando uma penalidade pequena por *step* faz com que ele procure por recompensas.

Quanto aos eventos que definem o fim do episódio são três:

- Chegar ao destino (R_d);
- Tempo se esgotar (P_{step});
- Tombar o veículo (P_t);

Se ao fim de um episódio, o agente chegar ao destino antes do tempo se esgotar, ele deve receber a recompensa máxima. Caso o tempo se esgote o agente deve receber uma punição. Em ambos os casos será considerado o número de *checkpoints* atingidos (o agente pode se direcionar até o destino ignorando-os), bem como as colisões com a parede e calçada. Desta forma, podemos avaliar a qualidade com que o agente conduziu o veículo. Por exemplo, se ele chegou ao destino se chocando com a calçada ou se fez um trajeto "limpo"; outro exemplo, no segundo cenário, o agente receberá uma recompensa maior caso chegue mais perto do destino do que se atingir nenhum *checkpoint*. Por fim, se o agente

tombar o veículo ele receberá a punição máxima, independente de seu comportamento antes do tal evento.

Tendo isso em vista, para formalização matemática dos episódios temos o seguinte:

- Chega ao destino: $R_T = R_d + R_c - P_c - P_p$
- Não chega a tempo: $R_T = R_c - P_{step} - P_c - P_p$
- Tomba o carro: $R_T = P_t$

Onde R_t é a recompensa final do episódio. $r_c \doteq \frac{R_c}{n_c}$, com n_c sendo o número de *checkpoints* no trajeto específico. Similarmente, temos $P_c \doteq p_c * n$ e $P_p \doteq p_p * n$, que são as punições totais (o produto da punição do evento pelo número de ocorrências n) pelas colisões com a calçada e a parede, respectivamente. Além disso, o agente possui 800 *steps* para atingir o destino. Este limite é aproximadamente 15 segundos, tempo suficiente dado que todos os trajetos são curtos. Com isso $P_{step} \doteq p_{step} * 800$.

As recompensas variam de -1 a 1, este intervalo não deve ser ultrapassado pois pode levar a instabilidade no treino. Como foi visto, algumas das recompensas/punições acima depende de outras. A tabela abaixo exibe o valor de todas as constantes de recompensas, estas que foram definidas e ajustadas pelo autor até gerar um treino mais estável.

Tabela 1 – Valor das recompensas e punições dos eventos.

Evento	Constante	Valor
Atingir todos *checkpoints*	R_c	0,8
Chegar ao destino	R_d	0,2
Colidir com a calçada	p_c	-0,05
Colidir com a parede	p_p	-0,05
Esgotar o tempo	P_{step}	-0,7
Tombar o veículo	P_t	-1

4.1.4 O veículo

Já vimos como funciona o agente, agora vamos destrinchar como funciona a condução do *GameObject* do veículo. O que foi explicado em O agente se trata dos componentes que estão associados ao objeto do topo da hierarquia, o *GameObject Car*. Abaixo do mesmo há dois objetos *body* e *spoiler* que são basicamente os *meshes 3D* que dão o visual do veículo.

O terceiro objeto é *Wheels* que agrega os objetos *Meshes* e *Wheel Colliders*, o primeiro é um conjunto dos *meshes 3D* das rodas, o segundo agrupa os *colliders* das mesmas. Estes últimos possuem um *component* chamado *Wheel Collider*, que é responsável pela física da roda (Technologies (2023)) podendo configurar o raio, massa, amortecimento,

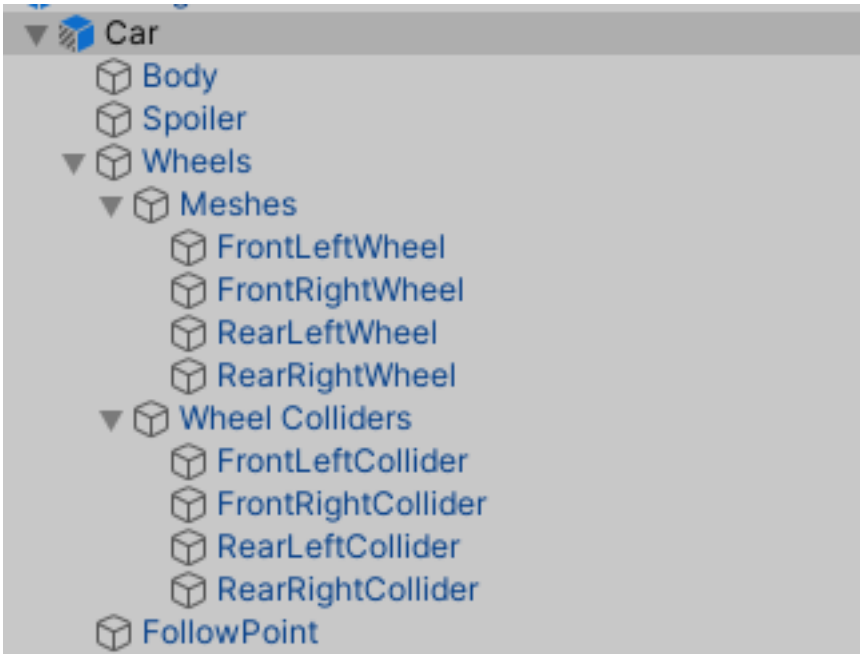

Figura 7 – Hierarquia dos *GameObjects* que compõe o veículo.

etc. Todos os atributos estão com o valor padrão, exceto o raio que foi alinhado com o do *mesh 3D*.

Para o controle do veículo, foi adaptado um código fonte disponível em (PrismYoutube (2022)), este código pega o input do usuário e atribui os mesmos a propriedades padrões dos *colliders* das rodas dianteiras, o input "vertical" é aplicado ao torque do motor (*motorTorque*), o "horizonta" ao ângulo de esterçamento do veículo (*steeringAngle*, limitado a 30°). Ao frear é aplicado uma *breakforce* nas quatro rodas. *breakforce*, *motorTorque* e *steeringAngle* são todos propriedades do componente *Wheel Collider* da Unity, basta associar os inputs do usuário a eles que a *engine* da Unity3D lida com a física do veículo.

Para o projeto, foi feita uma adaptação neste código-fonte, em vez de se aplicar o input do usuário, foi aplicado os valores que vem do *actionBuffer*, que é um argumento de *OnActionReceived()* a função que é responsável por tratar os dados das ações tomadas pelo agente. O *actionBuffer* é um array que contém em cada elemento os valores da política naquele estado, isto é, o "input do agente".

O agente no projeto pode ser entendido como o conjunto de *components* que fazem parte do GameObject pai, o "Car", seriam eles: behavior Parameters, o Car Agent, o Decision Requester e o Ray Perception sensor 3D.

4.2 Os algoritmos e hiperparâmetros

Para executar o treino é necessário especificar para o **ml-agents** o algoritmo e suas configurações, para isso existe um arquivo **.yaml** que é responsável por isso. Abaixo, um

exemplo seguido da explicação de cada parâmetro:

```
behaviors:
    MoveToTarget:
        trainer_type: ppo
        max_steps: 9600000
        time_horizon: 64
        summary_freq: 60000
        keep_checkpoints: 16
        checkpoint_interval: 600000
        hyperparameters:
            learning_rate: 1.0e-5
            batch_size: 1024
            buffer_size: 10240
            beta: 5.0e-2
            epsilon: 0.1
            lambd: 0.99
            num_epoch: 8
            learning_rate_schedule: linear
            beta_schedule: linear
            epsilon_schedule: linear
        network_settings:
            normalize: false
            hidden_units: 64
            num_layers: 2
        reward_signals:
            extrinsic:
                gamma: 0.99
                strength: 1.0
```

O arquivo começa com o **behaviors** que é uma lista de configurações dos comportamentos do agente, neste projeto haverá apenas um que é chamado **MoveToTarget**. Abaixo segue uma lista dos hiperparâmetros mais relevantes, o texto aqui é em grande parte traduzido da documentação oficial do **ml-agents** (Juliani et al. (2020)) com algumas remoções de informações não relevantes pra este projeto e com alguns adendos do autor caso necessário.

trainer_type: O algoritmo que será usado, neste projeto serão PPO ou SAC

max_steps: O número máximo de passos (observações coletadas e ações tomadas pelo agente) tomados no ambiente. No simulador a tarefa sempre será episódica, o tamanho

do episódio pode variar nos ajustes, mas cada um possui em torno de 1200 *steps*.

time_horizon: O número de *steps* anteriores a ser coletado por agente para adicionar ao *buffer* de experiência. Quando este limite é alcançado antes do fim de um episódio, um valor estimado é usado para prever a expectativa geral de recompensa a partir do estado atual do agente. Por isso, esse parâmetro varia entre menos enviesado mas com alta variância estimada (longo prazo) e mais enviesado mas com estimativa de menor variância (curto prazo). Em casos onde há frequente sinais de recompensas ou em episódios muito longos, um número menor pode ser mais ideal.

hyperparameters->learning_rate: Taxa inicial do salto a cada atualização do gradiente descendente. Este número geralmente deve diminuir com se o treino está instável e a recompensa com aumenta consistentemente.

hyperparameters->batch_size: O número de experiências coletadas para cada atualização do gradiente descendente. Em caso de usar ações contínuas ele deve estar na casa do milhares, caso contrário na casa das dezenas deve bastar.

hyperparameters->batch_size: Número de experiências a ser coletada antes de atualizar o modelo da política. Tipicamente, um buffer size maior corresponde a um treino mais estável. No caso do SAC este número deve ser milhares de vezes maior que um episódio típico, pois o algoritmo deve aprender de experiências velhas quanto mais novas.

hyperparameters->beta: (somente PPO) força da regularização da entropia, que faz com que a política seja "mais aleatória". Isto garente que o agente explore apropriadamente o espaço da ação durante o treino. Aumentá-lo faz com que o agente tome mais ações aleatória. Isto deve ser ajustado de modo que o a entropia (medido pelo tensorboard) lentamente decresça conforme aumenta a recompensa. Se a entropia cair muito rápido aumente o **beta**, caso demore demais diminua-o.

hyperparameters->epsilon: (somente PPO) Influencia no quão rapidamente a política pode evoluir durante o treino. Corresponde ao limite da divergência entre as velhas e novas políticas durante a atualização do gradiente descendente. Um valor menor leva a atualizações mais estáveis, mas deixa o processo de aprendizagem mais lento.

hyperparameters->lambd: (somente PPO) parâmetro de regularização (lambda) usado quando calculado o estimador de valor generalizado (GAE (Schulman et al. (2015))). Isto pode ser entendido em o quanto o agente depende no seu atual valor estimado quando atualizando o valor estimado. Valores baixos correspondem à apoiar-se mais no valor atual (alto viés/*bias*) e valores elevados corresponde a confiar mais nas recompensas recebidas pelo ambiente (que pode causar alta variância). (geralmente varia de 0,9 a 0,99)

hyperparameters->num_epoch: (somente PPO) O número de passagens a fazer pelo buffer_size quando otimizando gradiente descendente. Este deve crescer semelhante ao batch_size. Diminui-lo tende a ter atualizações mais estáveis, aumentá-lo tem-se um treino mais lento. (geralmente varia entre 3 a 10)

hyperparameters->learning_rate_schedule: Determina se o valor do learning_rate muda durante o treino, os valores podem ser *linear* ou *constant*. Sendo o primeiro ele irá decair linearmente até zero quanto executar o *max_steps*. O segundo mantém o valor constante. É recomendado adotar *linear* para PPO e o outro quando usar SAC.

hyperparameters->beta_schedule: (somente PPO) Similarmente ao acima mas para o **beta**.

hyperparameters->epsilon_schedule: (somente PPO) Similarmente ao acima mas para o **epsilon**.

hyperparameters->buffer_init_steps: (somente SAC) Número de experiências a coletar no buffer antes de atualizar o modelo da política. Inicialmente o buffer é preenchido com ações aleatórias, o que é muito útil para exploração. Este número deve estar na casa de dezenas de vezes maior que um típico episódio.

hyperparameters->init_entcoef: (somente SAC) Corresponde a entropia inicial definida no começo do treino e é o quanto o agente deve explorar inicialmente. No SAC, o agente é incentivado a tomar decisões aleatorias a fim de explorar melhor o ambiente. O coeficiente de entropia é ajustado automaticamente para um valor alvo predefinido então o *init_entcoef* é apenas o valor inicial. Quanto maior o valor maior a exploração inicial. Para ações contínuas o típico valor está no invervalo 0,5-1,0, discreto em 0,05-0,5.

network_settings->hidden_units: O número de nós em cada camada intermediária da rede neural totalmente conectada (valores típicos giram em torno de 32 a 512).

network_settings->num_layers: O número de camadas intermediárias na rede neural (tipicamente tem valor de 1 a 3).

network_settings->normalize: Se normalização deve ser aplicada no vetor de observação. Esta normalização pode melhorar o treino em caso de tarefas que lidam com controles contínuos complexos, mas pode atrapalhar caso contrário.

4.3 Desenvolvimento

O objetivo final é chegar a um agente que saiba percorrer todos os 17 trajetos de forma eficiente, para isso é preciso analisar as estatísticas produzidas pelo treino, saber

se a política se aproximou suficientemente do comportamento ótimo de modo que não cometa erros. Antes que se chegue a este nível é preciso que ele demonstre sucesso em treinos mais simples, então dividimos as etapas em 3 **desafios**. Os dois primeiros serão em trajetos específicos, mas com níveis diferentes de dificuldade. O último será o treino o desafio geral, onde ele deve percorrer qualquer rota dada a ele.

4.3.1 Treinamento e teste

O treinamento é executado via linha de comando, após sua conclusão é gerado um arquivo **.onnx** que é política final do agente, isto é, a rede neural com os coeficientes gerados pelo algoritmo durante o treino, o seu "cérebro". Finalmente podemos por o agente em teste, de volta ao editor da Unity3D, basta carregar o cérebro no agente e executar o teste, o veículo agora se move de acordo como a política traduz os estados em ações, definida no arquivo .onnx, sem qualquer input do usuário, com o agente agindo de acordo com os resultados do treino, seja ele um sucesso ou não.

Quando o treino é finalizado, suas estatísticas são exibidas pelo **tensorboard**, um pacote Python incluso no pacote ml-agents. Nelas é possível ter uma visualização do desempenho do agente durante o treino e sua curva de aprendizado. Quais são e o que significam estão na documentação do ML-agents em (Juliani et al. (2020)). A estatística que iremos analisar aqui será o gráfico de **recompensa acumulada** (cumulative reward) que exibe a recompensa mediana por episódio, isso nos fará entender se o agente convergiu e com que velocidade. As demais estatísticas não tem muita utilidade para o nosso propósito, elas são mais úteis para saber como ajustar os hiperparâmetros em caso de um treino não sucedido.

Nos testes o agente deverá percorrer os mesmos trajetos nos quais treinou, como as observações do agente não incluem qualquer informação sobre qual trajeto ele está percorrendo no momento, o autor não viu necessidade de separar os trajetos em um conjuntos de treino e teste, porém isso pode ser explorado em estudos futuros.

O primeiro desafio inclui dois treinos um em cada trajeto fácil diferente, por fácil entende-se uma rota que seja necessário fazer apenas uma curva. O segundo desafio inclui três treinos em rotas com duas ou mais curvas. Por fim o desafio geral, onde ele treinará em todos os dezessete trajetos. Após obtermos o "cérebro" gerado, este será colocado em teste percorrendo o mesmo trajeto o qual treinou (no caso dos desafios específicos), percorrendo-os dez vezes. No caso do desafio geral, 3 vezes em cada rota.

O consolidado dos desafios está resumido na Tabela 2, a segunda coluna "Rotas" se refere ao nome dado ao *GameObject* da rota dentro do editor Unity.

Tabela 2 – Resumo dos desafios

Desafio	Rotas	Qtde. testes
Específico em trajetos fáceis	*Path(8)* *Path(0)*	10 em cada rota
Específico em trajetos difíceis	*Path(3)* *Path(5)* *Path(6)*	10 em cada rota
Geral	Todas as 17 rotas	51 (3 em cada)

4.3.2 Análise

Para cada treino será testado três algoritmos: PPO, BC e GAIL. Então o primeiro é um treino/teste com apenas RL, enquanto os outros dois se tratam de Aprendizado por Imitação. A análise de desempenho avaliará os seguintes quesitos:

- Se convergiu ou não;
- Mediana final de recompensa por episódio;
- Porcentagem de conclusão da rota nos testes (cada rota no geral);
- Média de recompensa na rota (caso treino em rota especifíca);
- Média de recompensa por rota (caso geral).

Ao fim devemos ser capazes de responder as seguintes perguntas:

- Foi possível ensinar um agente a percorrer trajetos?
- Qual algoritmo mais eficiente nos testes?
- Qual algoritmo convergiu mais rápido nos treinos?

Parte III

Parte Final

5 Resultados e Discussão

5.1 Desafio em trajeto específico dificuldade fácil

Aqui o treino foi feito em dois trajetos (identificados por *Path(0)* e *Path(8)* no projeto), ambos exigem que o agente faça uma conversão do veículo (ver Figura 8). O primeiro exige que o agente faça uma curva à esquerda e siga reto até o destino, o outro exige o agente siga reto e faça uma conversão à direita.

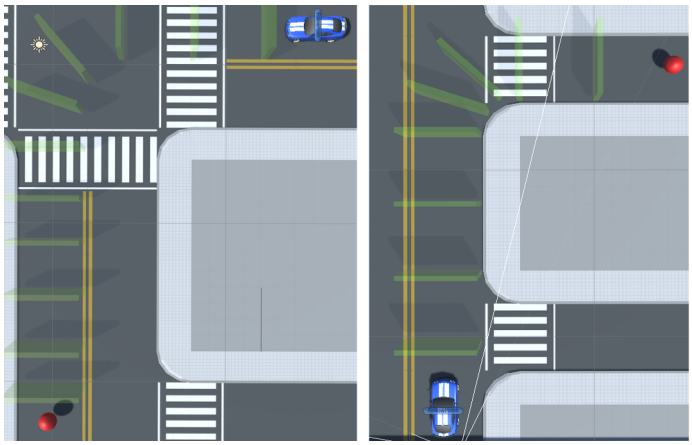

Figura 8 – Trajetos deste desafio, à esquerda o *Path(0)* e à direita o *Path(8)*.

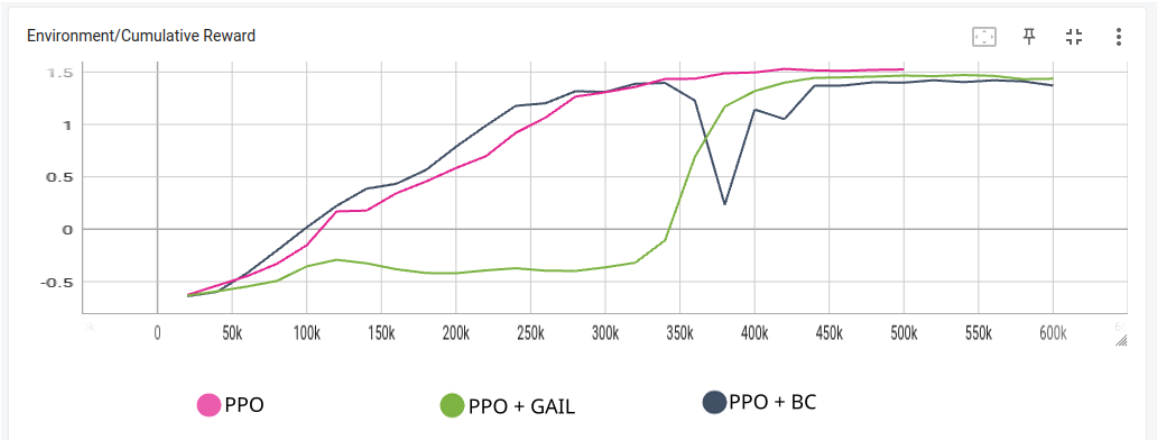

Figura 9 – Recompensa cumulativa mediana por *steps* durante o treino do agente no Path(0).

O treinamento não encontrou problemas para convergir neste desafio em qualquer dos três algoritmos. PPO foi o que obteve a melhor mediana final no *Path(0)* mesmo dispondo de menos *steps* para treinar (identificado pela linha rosa nas Figuras 9 e 10). Embora tenham convergido no *Path(0)*, GAIL e BC se mostraram instáveis. O primeiro demorou muito para mostrar algum progresso na recompensa, enquanto BC apresentou

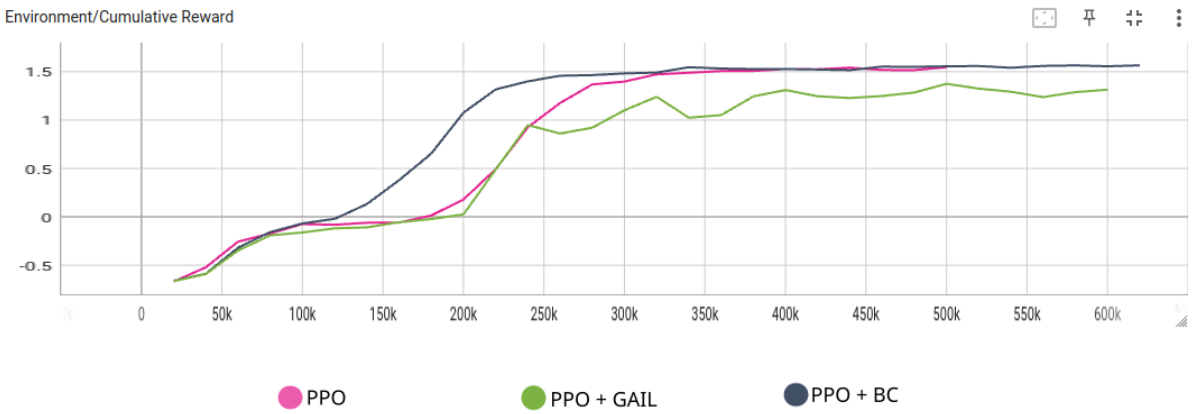

Figura 10 – Recompensa cumulativa mediana por *steps* durante o treino do agente no *Path(8)*.

uma anomalia após 350 mil *steps* voltando a se estabilizar após 450 mil passos, como mostra a Figura 9. Quanto ao *Path(8)*, o treino que utilizou BC convergiu mais rápido que o de PPO e também obteve uma mediana maior que este. Neste trajeto o GAIL foi o mais instável, o único que não convergiu (linha verde da Figura 10).

Tabela 3 – Consolidado de resultados por rota e algoritmo, contém a mediana final da recompensa no treino, a coluna de desempenho teste é quantas rotas completou de quantas tentativas e média de recompensas nos testes.

Rota	Algoritmo	Mediana final rec. Treino	Desempenho teste	Média rec. Testes
Path(0)	PPO	1.524	10/10	1
	GAIL	1.436	10/10	1
	BC	1.371	10/10	1
Path(8)	PPO	1.546	10/10	1
	GAIL	1.314	6/10	0.585
	BC	1.565	10/10	1

Nos testes, com exceção do algoritmo GAIL no *Path(8)*, todos os agentes desempenharam a tarefa sem falha alguma. O cérebro treinado com o algoritmo BC subiu na calçada uma vez em cada teste, mas a punição por uma ocorrência não é suficiente para afetar sua média. O agente treinado com GAIL no *Path(8)* falhou em 4 dos 10 testes, não há certeza do porque disto, mas talvez seu desempenho foi afetado porque a rota 8 exige que a curva seja feita após percorrer um trajeto reto, o contrário do *Path(0)*, que exige que a curva seja feita logo após o ponto de partida. Esta exigência de mudança no comportamento do agente pode ser a causa do GAIL ter falhado.

5.2 Desafio em trajeto específico difícil

Para este desafio foi testado em três rotas diferentes: *Path(3), Path(5) e Path(6)*. A Figura 11 exibe a visão superior dos três trajetos. O primeiro é o mais complexo possuindo

três curvas alternadas, enquanto *Path(5)* requer que o carro faça duas conversões à direita e o último trajeto contém duas curvas alternadas.

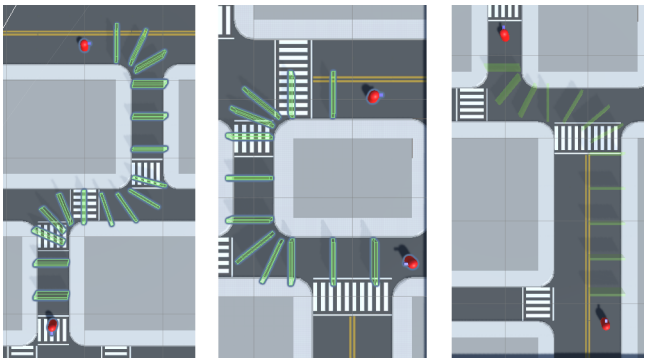

Figura 11 – Visão superior das três rotas do desafio específico difícil.

Neste desafio, a diferença do resultado dos treinos entre os algoritmos aumentou. Na Figura 12 pode-se notar que apenas o PPO foi estável e convergiu no *Path(3)*, enquanto BC a partir da metade do treino passou a perder a recompensa, ficando preso em um mínimo local. O mesmo ocorreu com GAIL, porém este não chegou a apresentar progresso em momento algum.

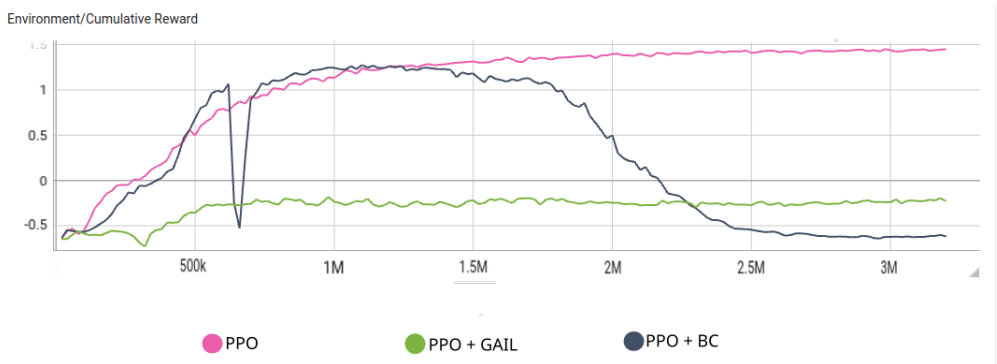

Figura 12 – Recompensa cumulativa mediana por *steps* durante o treino do agente no *Path(3)*.

No *Path(5)*, que exige duas curvas à esquerda, PPO e BC convergiram rapidamente, com BC se mantendo estável desta vez. GAIL, por outro lado, ficou preso em um mínimo local, mesmo dispondo de mais *steps* que os outros, não conseguiu apresentar melhora (ver Figura 13).

No *Path(6)*, os três algoritmos apresentaram uma melhora constante e estável nos treinos. Na Figura 14, é possível ver que apesar de o PPO ter obtido uma melhor mediana final (linha rosa na Figura), o BC teve um aprendizado mais rápido inicialmente (linha preta). Neste trajeto, GAIL também se mostrou mais instável que os demais, porém obteve um desempenho melhor aqui que noutros trajetos, inclusive com uma mediana final ligeiramente acima de BC no *Path(6)*.

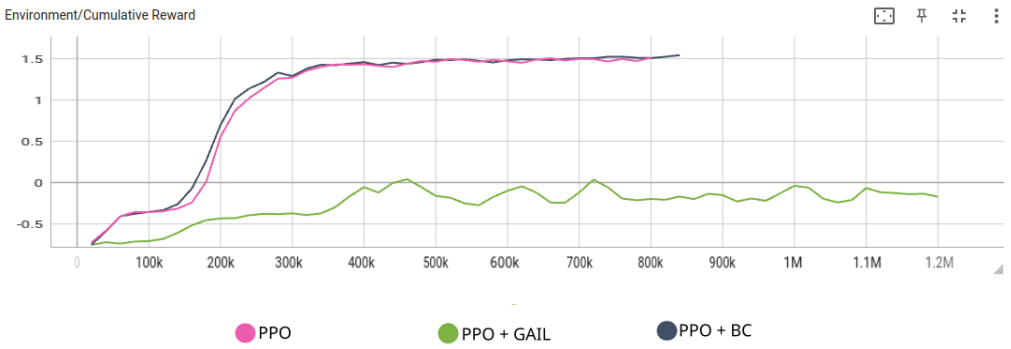

Figura 13 – Recompensa cumulativa mediana por *steps* durante o treino do agente no *Path(5)*.

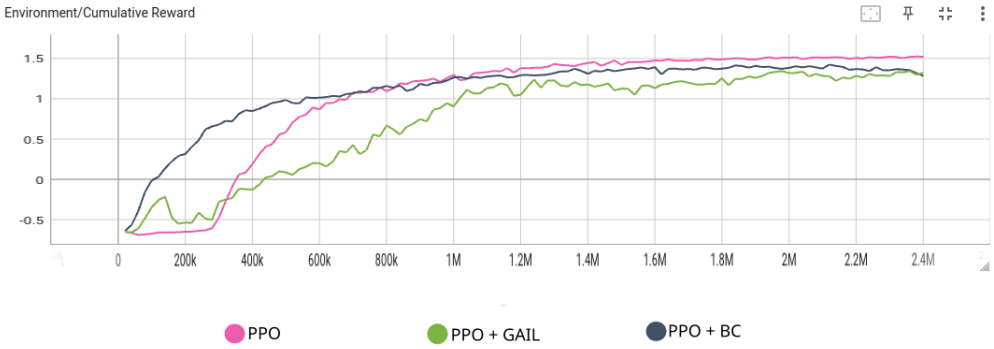

Figura 14 – Recompensa cumulativa mediana por *steps* durante o treino do agente no *Path(6)*.

Tabela 4 – Consolidado de resultados por rota e algoritmo do segundo desafio.

Rota	Algoritmo	Mediana final rec. Treino	Desempenho testes	Média rec. Testes
Path(3)	PPO	1,449	10/10	0,97
	GAIL	-0,229	0/10	-0,261
	BC	-0,618	0/10	-0.693
Path(5)	PPO	1,509	10/10	0,98
	GAIL	-0,171	0/10	-0,252
	BC	1,505	10/10	1
Path(6)	PPO	1,522	10/10	0,993
	GAIL	1,323	10/10	0,99
	BC	1,284	5/10	0,42

Todos desempenharam conforme o treino durante os testes com exceção do BC no *Path(6)* concluiu apenas metade das rotas. Durante o teste do *Path(6)* foi observado uma diferença interessante entre o agente treinado com PPO que usa apenas RL e o GAIL que se utiliza de IL: o primeiro tendia a explorar de uma falha de detecção de colisão do veículo com a calçada poupando assim um pouco de tempo (e de punição por *step*) em vez de seguir na via. Isso ocorre porque da forma que o agente passa sobre a calçada o ambiente raramente detecta uma colisão ali, com o agente treinado em GAIL isto não ocorre, já que o mesmo tenta imitar o comportamento na demonstração feita por um humano que seguiu

na via normalmente.

Na Tabela 4 pode-se perceber que o PPO concluiu todas as rotas em todas as tentativas, porém sua recompensa média não atingiu o máximo, sempre cometendo alguma infração por tentativa. BC só obteve um desempenho satisfatório no *Path(5)*, inclusive com recompensa máxima, maior que o PPO, porém falhou nos demais trajetos. Falhou em metade dos testes na última rota e não concluiu teste algum, no *Path(3)*. Por fim, GAIL não concluiu trajeto algum exceto no teste do *Path(6)*, onde chegou ao destino em todas as tentativas.

5.3 Desafio em todos os trajetos

Finalmente, no último desafio temos o PPO e BC convergindo relativamente rápido enquanto GAIL com um treino muito instável mesmo dispondo do dobro de *steps* (Figura 15). Por isso, neste desafio houve uma discrepância muito grande entre GAIL e os demais, mesmo após diversos ajustes nos hiperparâmetros e isto refletiu nos testes onde obteve uma recompensa média negativa.

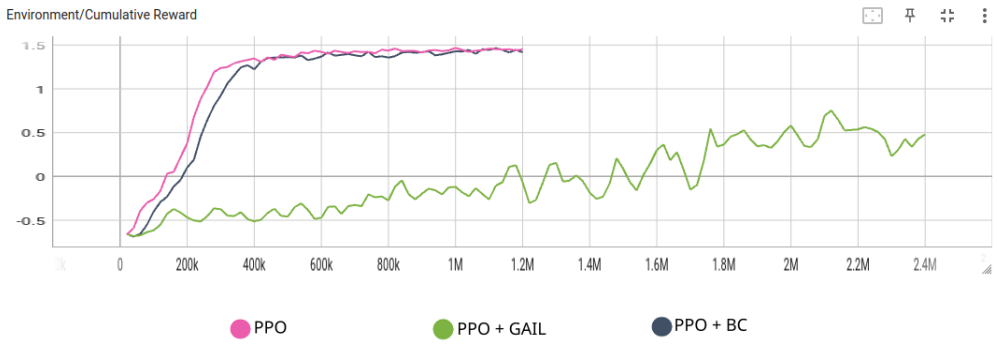

Figura 15 – Recompensa cumulativa mediana por *steps* durante o treino do agente em todos os trajetos.

Nas tabelas 5, 6 e 7 pode-se ver o detalhamento dos testes em cada rota dos algoritmos PPO, GAIL e BC, respectivamente. O *Path(10)* mostrou-se como o trajeto mais difícil para PPO e BC, com a menor recompensa média nestes algoritmos e a segunda menor média no GAIL.

Interessante ressaltar que BC nestes testes conseguiu concluir o *Path(3)* em todas as tentativas em contraste com o desafio anterior onde falhou em todas as dez tentativas, o que nos leva a crer que treinar em outros trajetos ajude o agente a cumprir um trajeto específico em vez de treinar somente em um único trajeto, pelo menos utilizando-se de Aprendizado por Imitação.

Por fim na Tabela 8, temos o resultado consolidado geral dos testes deste desafio, onde é possível ver uma diferença muito grande entre GAIL e os demais. GAIL cumpriu

Tabela 5 – Detalhamento do desempenho no teste do desafio geral de PPO, inclui todas as tentativas, suas recompensas e a recompensa média por rota.

Rota	Tentativa 1		Tentativa 2		Tentativa 3		Rec. Média
	Concluiu?	Rec.	Concluiu?	Rec.	Concluiu?	Rec.	
Path(0)	sim	1	sim	1	sim	1	1
Path(1)	sim	1	sim	1	sim	1	1
Path(2)	sim	1	sim	1	sim	1	1
Path(3)	não	-0,18	sim	0,85	sim	1	0,56
Path(4)	sim	1	sim	1	sim	1	1
Path(5)	sim	1	sim	0,8	sim	1	0,93
Path(6)	sim	1	não	-0,31	sim	1	0,56
Path(7)	sim	0,85	sim	1	sim	0,9	0,91
Path(8)	sim	1	sim	0,9	sim	1	0,96
Path(9)	sim	1	sim	0,95	sim	1	0,98
Path(10)	não	0,12	não	0,02	sim	1	0,38
Path(11)	sim	1	sim	1	sim	1	1
Path(12)	sim	1	sim	1	sim	1	1
Path(13)	sim	1	sim	1	sim	1	1
Path(14)	sim	1	sim	1	sim	1	1
Path(15)	sim	1	sim	1	sim	1	1
Path(16)	sim	1	sim	1	sim	1	1

Tabela 6 – Detalhamento do desempenho no teste do desafio geral de GAIL.

Rota	Tentativa 1		Tentativa 2		Tentativa 3		Rec. Média
	Concluiu?	Rec.	Concluiu?	Rec.	Concluiu?	Rec.	
Path(0)	sim	1	não	-0,677	sim	-0,402	-0,026
Path(1)	não	0,231	não	-0,01	não	-0,253	-0,011
Path(2)	não	-0,375	não	-0,153	não	-0,551	-0,36
Path(3)	não	-0.412	não	0,002	não	-0,588	-0,333
Path(4)	não	-0,086	sim	1	não	-0,504	0,137
Path(5)	não	-0,278	não	0,132	não	-0,515	-0,22
Path(6)	sim	1	sim	1	não	0,201	0,734
Path(7)	não	-0,297	não	-0,22	não	-0,636	-0,384
Path(8)	não	0,087	não	-0,087	não	0,215	0,072
Path(9)	não	-0,084	não	0,204	não	-0,104	0,006
Path(10)	não	-0,44	não	-0,677	não	-0,586	-0,566
Path(11)	não	0,063	não	-0,182	sim	1	0,293
Path(12)	não	0,074	não	-0,44	não	-0,655	-0,292
Path(13)	nào	0,191	não	-0,686	não	-0,668	-0,718
Path(14)	não	0,098	não	-0,644	não	-0,553	-0,366
Path(15)	não	-0,579	não	-0,397	não	-0,47	-0,604
Path(16)	sim	1	não	-0,665	não	-0,653	-0,106

Tabela 7 – Detalhamento do desempenho no teste do desafio geral de BC.

Rota	Tentativa 1		Tentativa 2		Tentativa 3		Rec. Média
	Concluiu?	Rec.	Concluiu?	Rec.	Concluiu?	Rec.	
Path(0)	sim	1	sim	1	sim	1	1
Path(1)	sim	1	sim	1	sim	1	1
Path(2)	sim	1	sim	1	sim	1	1
Path(3)	sim	1	sim	0,9	sim	1	0.967
Path(4)	sim	1	sim	1	sim	1	1
Path(5)	sim	1	sim	1	sim	1	1
Path(6)	sim	1	sim	1	sim	1	1
Path(7)	sim	1	sim	0,95	sim	1	0,983
Path(8)	sim	1	sim	0,95	sim	1	0,983
Path(9)	sim	1	sim	1	sim	1	1
Path(10)	não	0,224	não	0,158	não	0,221	0,201
Path(11)	sim	1	sim	1	sim	1	1
Path(12)	sim	1	sim	1	não	-0,41	0,53
Path(13)	sim	1	sim	1	sim	1	1
Path(14)	sim	1	sim	1	sim	1	1
Path(15)	sim	1	sim	1	não	0,12	0,707
Path(16)	sim	1	sim	1	sim	1	1

apenas 7 dos 51 testes (aprox. 13,7%), enquanto PPO e BC concluíram 47(92,1%) e 46(90,2%) dos 51 testes, respectivamente.

Tabela 8 – Consolidado de resultados no desafio geral.

Rota	Algoritmo	Mediana final rec. Treino	Desempenho testes	Média rec. Testes
Todas	PPO	1,451	47/51	0,899
	GAIL	0,479	7/51	-0,161
	BC	1,416	46/51	0,904

Conclusões e Trabalhos Futuros

Ao longo deste projeto, foi perceptível a dificuldade de se criar uma IA para veículo autônomo apesar de a duração dos treinos ter sido rápida (o desafio geral com PPO e BC convergindo com cerca de 90 minutos), vale lembrar que este simulador trata-se de um modelo excessivamente simples. Por outro lado, o agente foi bastante eficiente em realizar sua tarefa durante os testes após os treinos. Ficou notada a vantagem do Aprendizado por Reforço com PPO em relação ao Aprendizado por Imitação com BC e GAIL, principalmente este último que se mostrou bastante instável. Este projeto conseguiu cumprir o objetivo de provar ser possível criar um ambiente de treinamento para um agente capaz de conduzir um automóvel.

Trabalhos Futuros

Podemos dividir os estudos futuros em três frentes: melhorias no modelo e no agente e exploração de novos algoritmos.

Melhorias no modelo e simulação

Algumas das melhorias mais óbvias que podem ser feitas são a adição de outros elementos estáticos no cenário que estão ausentes nesta versão, por exemplo, semáforos, postes, obstruções na pista, prédios, etc.

Como foi dito, este modelo atualmente é muito simples, o desafio de se criar um agente completamente capaz de conduzir um veículo em ruas públicas é um caminho longo e envolve muitos dilemas éticos. Felizmente, a Unity3D fornece uma gama de ferramentas que facilita a criação de cenários hipotéticos para testar o agente. Por exemplo, treinar um cenário multi-agente, onde poderíamos ter dentro da mesma simulação mais de um agente circulando nas ruas, além de outros veículos não inteligentes dirigidos por humanos. A simulação com pedestres talvez seja a mais importante de se simular, pois, um dos maiores temores que se tem com a tecnologia de veículos autônomos é como ela reagiria em um acidente iminente. Além do fato de que um condutor que esteja sempre atento e não cometa erros típicos humanos é um dos grandes objetivos da criação de carros autônomos.

Melhorias no agente

Aqui foi testado um agente que consiga percorrer trajetos extremamente curtos, o agente tinha um número máximo de 800 *steps* por rota para se chegar ao destino, o que dura em torno de 15 segundos. Este limite foi colocado para evitar que o veículo ficasse

imóvel e o algoritmo preso em um mínimo local, porém, seria interessante aumentar o limite de *steps* com o tamanho de trajetos, trajetos longos que demorassem pelo menos alguns minutos poderiam gerar um novo caso de estudos.

Outro problema com o agente foi a falta de uso de processamento de imagem, ele dispõe apenas de sensores de distância que conseguem detectar alguns elementos como *checkpoint* e calçada. Com o uso de imagem o agente teria além de mais informação ele a disposição as mesmas observações que um humano possui enquanto dirige.

Exploração de novos algoritmos

Como foi visto nos resultados, GAIL teve treinos instáveis não convergindo algumas vezes além de ter desempenhado pior nos testes, por outro lado, isso não descarta totalmente o algoritmo. Estes problemas podem ter sido causado por sensibilidade nos hiperparâmetros e uma exploração maior deles pode resultar em um treino mais estável e testes mais bem sucedidos. Como foi mencionado na fundamentação teórica, o pacote ML-Agents da Unity possui SAC como um dos algoritmos disponíveis, sendo que este foi criado justamente para ser menos sensível aos hiperparâmetros comparado a outros métodos, uma combinação SAC+GAIL pode gerar resultados melhores dos que foram vistos aqui.

Referências

ANDRADE, A. Game engines: a survey. *EAI Endorsed Transactions on Game-Based Learning*, European Alliance for Innovation n.o., v. 2, n. 6, p. 150615, nov. 2015. Disponível em: <https://doi.org/10.4108/eai.5-11-2015.150615>. Citado na página 6.

BHATTACHARYYA, R. et al. Modeling human driving behavior through generative adversarial imitation learning. 2020. Citado na página 15.

DOSOVITSKIY, A. et al. CARLA: An open urban driving simulator. In: LEVINE, S.; VANHOUCKE, V.; GOLDBERG, K. (Ed.). *Proceedings of the 1st Annual Conference on Robot Learning*. PMLR, 2017. (Proceedings of Machine Learning Research, v. 78), p. 1–16. Disponível em: <https://proceedings.mlr.press/v78/dosovitskiy17a.html>. Citado na página 17.

HAARNOJA, T. et al. *Soft Actor-Critic Algorithms and Applications*. 2018. Citado na página 14.

HO, J.; ERMON, S. Generative adversarial imitation learning. In: LEE, D. et al. (Ed.). *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2016. v. 29. Disponível em: <https://proceedings.neurips.cc/paper_files/paper/2016/file/cc7e2b878868cbae992d1fb743995d8f-Paper.pdf>. Citado na página 15.

HUSSEIN, A. et al. Imitation learning: A survey of learning methods. *ACM Computing Surveys*, Association for Computing Machinery (ACM), v. 50, n. 2, p. 1–35, abr. 2017. ISSN 1557-7341. Disponível em: <http://dx.doi.org/10.1145/3054912>. Citado na página 15.

ITCH.IO. *Most used Engines*. [S.l.], 2023. Disponível em: <https://itch.io/game-development/engines/most-projects>. Citado na página 7.

JULIANI, A. et al. Unity: A general platform for intelligent agents. *arXiv preprint arXiv:1809.02627*, 2020. Disponível em: <https://arxiv.org/pdf/1809.02627.pdf>. Citado 2 vezes nas páginas 29 e 32.

KELLEHER, J. D.; NAMEE, B. M.; D'ARCY, A. *Fundamentals of machine learning for predictive data analytics*. London, England: MIT Press, 2015. (The MIT Press). Citado na página 9.

MÉXAS, R. P. ComparaÇÃo do desempenho de algoritmos de aprendizado de mÁquina por reforÇo e por imitaÇÃo na simulaÇÃo de veleiros autÔnomos. *Universidade Federal Fluminense*, 2021. Citado na página 18.

PADEN, B. et al. A survey of motion planning and control techniques for self-driving urban vehicles. *IEEE Transactions on Intelligent Vehicles*, Institute of Electrical and Electronics Engineers (IEEE), v. 1, n. 1, p. 33–55, mar. 2016. Disponível em: <https://doi.org/10.1109/tiv.2016.2578706>. Citado na página 5.

POMERLEAU, D. A. Efficient training of artificial neural networks for autonomous navigation. *Neural Computation*, MIT Press - Journals, v. 3, n. 1, p. 88–97, fev. 1991.

ISSN 1530-888X. Disponível em: <http://dx.doi.org/10.1162/neco.1991.3.1.88>. Citado na página 17.

PRISMYOUTUBE. *PrismYoutube/Unity-car-controller*. 2022. Disponível em: <https://github.com/PrismYoutube/Unity-Car-Controller>. Citado na página 28.

RUSSELL, S. Learning agents for uncertain environments (extended abstract). In: *Proceedings of the eleventh annual conference on Computational learning theory*. ACM, 1998. (COLT98). Disponível em: <http://dx.doi.org/10.1145/279943.279964>. Citado na página 15.

RUSSELL, S.; NORVIG, P. *Artificial intelligence*. 3. ed. Upper Saddle River, NJ: Pearson, 2009. Citado na página 8.

SAE INTERNATIONAL. *Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*. [S.l.], 2014. Disponível em: <https://www.sae.org/standards/content/j3016_202104/>. Citado 2 vezes nas páginas 1 e 5.

SCHULMAN, J. et al. *High-Dimensional Continuous Control Using Generalized Advantage Estimation*. 2015. Citado na página 30.

SCHULMAN, J. et al. *Proximal policy optimization algorithms*. 2017. Disponível em: <https://arxiv.org/abs/1707.06347>. Citado na página 14.

SOUZA, F. R. d. Aprendizado por reforço assistido por imitação para jogos digitais. *Universidade Federal de Juiz de Fora*, 2023. Citado na página 18.

STEAMDB. *What are games built with and what technologies do they use?* [S.l.], 2023. Disponível em: <https://steamdb.info/tech/>. Citado na página 7.

SUTTON, R. S.; BARTO, A. G. *Reinforcement Learning*. 2. ed. [S.l.]: MIT Press, 2018. Citado 3 vezes nas páginas 9, 10 e 12.

TECHNOLOGIES, U. *Wheel collider component reference*. 2023. Disponível em: <https://docs.unity3d.com/Manual/class-WheelCollider.html>. Citado na página 27.

UNITY ASSET STORE. *ARCADE: FREE Racing Car*. [S.l.], 2023. Disponível em: <https://assetstore.unity.com/packages/3d/vehicles/land/arcade-free-racing-car-161085>. Citado na página 21.

UNITY ASSET STORE. *CITY package*. [S.l.], 2023. Disponível em: <https://assetstore.unity.com/packages/3d/environments/urban/city-package-107224>. Citado na página 21.

URREA, C.; GARRIDO, F.; KERN, J. Design and implementation of intelligent agent training systems for virtual vehicles. *Sensors*, v. 21, n. 2, 2021. ISSN 1424-8220. Disponível em: <https://www.mdpi.com/1424-8220/21/2/492>. Citado na página 17.

WYMANN, B. et al. *TORCS, The Open Racing Car Simulator, v1.3.5*. 2013. <http://www.torcs.org>. Citado na página 17.

Apêndices

APÊNDICE A – Sobre os testes feitos e como reproduzi-los

Os testes dos quais a seção de resultados se refere estão disponíveis em vídeo nesta URL do Google Drive: https://drive.google.com/drive/folders/1WcdHMeFQLfDubvYvLJnqbobnysT5pJoq?usp=sharing.

Como reproduzir os testes está descrito na página do repositório oficial do projeto (https://github.com/antunesvitor/SimuladorDeConducao), porém de modo a deixar este texto completo com toda a informação necessária as instruções serão disponíveis aqui também.

Requisitos

- Sistema Operacional: Windows 11[1] ou Linux[2]
- Unity Hub
- Unity3D editor versão 2022.3.13f1[3]
- Git (opcional)

[1] A Unity3D está disponível para Windows 10 também, mas este projeto não foi testado nele, provavelmente é possível reproduzir/treinar sem problema algum, no Win10.

[2] As versões de Linux suportadas oficialmente é O Ubuntu 16.04, 18.04 e CentOS 7. O desenvolvimento do projeto foi feito no Arch Linux que NÃO é oficialmente suportado.

[3] Pode ser possível executar em outras versões 2022.3.x.

Como reproduzir

Aqui é explicado como reproduzir os testes feitos no projeto. Eles não obterão o exato resultado mostrado no projeto dado a natureza estocástica do agente e excesso de possíveis estados, porém um desempenho muito próximo é esperado.

Instalando e abrindo o projeto

1. Primeiro baixe e instale o Unity Hub.

- Windows: baixe e instale pelo site oficial da Unity (https://unity.com/download).
- Arch Linux: instale o pacote na AUR (https://aur.archlinux.org/packages/unity hub). Para isso terá que usar um gerenciador de pacotes da AUR como o yay (https://github.com/Jguer/yay) ou o paru (https://github.com/morganamilo/paru).
- Para as distribuições Linux oficialmente suportadas siga as instruções na documentação (https://docs.unity3d.com/2020.1/Documentation/Manual/GettingStartedInstallingHub.html.

2. Após isso clone ou baixe o repositório do projeto (https://github.com/antunesvitor/SimuladorDeConducao).

3. Abra o Unity Hub, vá em "Open" e navegue até o diretório onde você baixou o projeto e clique abrir. É possível que apareça um aviso de que a versão do editor não está instalada no seu computador, para isso existem duas opções:

   - (Recomendado) Baixar a versão exata do projeto: vá à página de arquivo de editores da Unity (https://unity.com/releases/editor/archive) clique na aba "Unity 2022.X" e clique no botão "Unity Hub" na versão 2022.3.7;
   - (Não recomendado) Alterar a versão do projeto: quando o aviso aparecer clique em "Choose another editor version" e então em "Install Editor Version" ele te apresentará as versões LTS disponíveis para download;

4. Após isso basta clicar duas vezes para abrir o projeto.

Testando os modelos

Para testar os modelos apropriadamente, é preciso testar um por vez e ajustar o projeto de acordo, alguns dos "cérebros" (arquivos .onnx na pasta Brains são a política definida pelo treino, a rede neural) são de trajetos específicos.

1. Selecione o objeto Car na hierarquia do projeto.

2. Arraste o "cérebro" que deseja treinar para o campo *Model* no inspetor à esquerda do editor.

3. Desabilite todos os *paths* que não for usar (consultar Nomenclatura dos cérebros).

4. Agora basta apertar o botão *play* no topo ao centro e o veículo já estará sendo conduzido pelo cérebro. (Obs. garanta que os campos *Behaviour Type* e *Inference Device* estejam ambos em *Default* ou *Inference Only* e *CPU* respectivamente, caso contrário não funcionará).

Nomenclatura dos cérebros

Os "cérebros" estão na pasta Assets/Brains e seguem o seguinte padrão de nome:

CarSim<algoritmo><versão>_<path>-<id>

<algoritmo> serve para identificar com qual algoritmo foi treinado. Possíveis valores são: PPO, SAC, PPO+BC, PPO+GAIL, PPO+GAIL+BC, etc.

<versão> é um número que identifica a versão do veículo, durante o projeto diversas versões foram criadas, atualmente só há uma versão dele (3.1), porém esta versão foi posta para deixar claro que possa haver mais de um agente em versões diferentes no mesmo ambiente.

<path> a rota a qual aquele cérebro treinou, se habilitar uma rota diferente do cérebro ele provavelmente não irá desempenhar bem a tarefa. O nome é igual aos GameObjects localizados abaixo do GameObject SpawnPointManager. O "GERAL" significa que ele treinou em todas as rotas.

<id> opcional, um identificador a mais para diferenciar um cérebro de outro caso os valores acima sejam iguais.