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

Análise do desenvolvimento de uma inteligência artificial condutora de um veículo

Vítor Guilherme Antunes

Santo André - SP, Dezembro de 2023

Vítor Guilherme Antunes

Análise do desenvolvimento de uma inteligência artificial condutora de um veículo

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

Vítor Guilherme Antunes

Análise do desenvolvimento de uma inteligência artificial condutora de um veículo/ Vítor Guilherme Antunes. – Santo André - SP, Dezembro de 2023-

59 p. : il. (algumas color.) ; 30 cm.

Orientador: Fernando Teubl Ferreira

Dissertação (Mestrado) – Universidade Federal do ABC – UFABC

Centro de Matemática, Computação e Cognição

Projeto de Graduação em Computação, Dezembro de 2023.

1. Palavra-chave1. 2. Palavra-chave2. I. Orientador. II. Universidade xxx. III. Faculdade de xxx. IV. Título

CDU 02:141:005.7

Agradecimentos

Agradeço a Xuxa, meus pais, cachorro, gato e papagaio, por ...

Agradeço ao meu orientador, XXXXXXXXX, por todos os conselhos, pela paciência e ajuda nesse período.

Aos meus amigos ...

Aos professores ...

À XXXXXX pelo apoio financeiro para realização deste trabalho de pesquisa.

"Não sei o que,
não sei o que,
não sei o que lá."
(Autor Desconhecido)

Resumo

Segundo a ABNT, o resumo deve ressaltar o objetivo, o método, os resultados e as conclusões do documento. A ordem e a extensão destes itens dependem do tipo de resumo (informativo ou indicativo) e do tratamento que cada item recebe no documento original. O resumo deve ser precedido da referência do documento, com exceção do resumo inserido no próprio documento. Umas 10 linhas (. . .) As palavras-chave devem figurar logo abaixo do resumo, antecedidas da expressão Palavras-chave:, separadas entre si por ponto e finalizadas também por ponto.

Palavras-chaves: latex. abntex. editoração de texto.

Abstract

This is the english abstract.

Keywords: latex. abntex. text editoration.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

IA	Inteligência Artificial
AS	Aprendizado Supervisionado
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

Introdução

No passado eram 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 que tomam decisões que controlam parcialmente o veículo, visando segurança e conforto ao condutor.

Veículos que possuem as assistências mencionadas acima comumente são chamados de "autônomos", porém neste artigo trataremos por "carro autônomo"um veículo que seja capaz de conduzir-se sem depender de um humano. De acordo com o padrão SAE J3016 a autonomia de veicular é dividida em níveis que vão do zero ao 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 de leitura, música ou compra são alguns exemplos de diversos outros que até então estavam apenas disponíveis em lugares específicos com laboratórios de pesquisa, mas agora já são amplamente aplicadas.

Existem três paradigmas de aprendizado de máquina: **aprendizado supervisionado**, **não supervisionado** e **aprendizado por reforço**. Este último aborda aprendizado que envolvam exercer uma atividade, nele, o **agente** durante seu treino, aprimora uma tarefa após várias tentativas e erro onde é premiado quando age corretamente e penalizado caso contrário. Este artigo introduzirá os dois primeiros paradigmas e discorrerá mais detalhadamente sobre o **AR**. Embora o **AS** também é necessário para a tarefa, ele é mais usado em reconhecimento de elementos no ambiente, algo que está fora do escopo deste projeto.

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

Este artigo propõe-se a apresentar um estudo do que é necessário para criar um agente condutor de um veículo dentro de um simulador que saiba percorrer um dado

trajeto em um ambiente urbano. Será analisado como modelar um simulador, como quais sensores o veículo deve possuir, quais observações deve fazer do ambiente para evitar colisões, entre outros.

Justificativa

Uma das maiores causas de morte no Brasil é por conta de acidentes de trânsito, chegando a um total de 43 mil óbitos em um único ano (CARVALHO, 2016), é sabido também que grande causa dessas mortes é por falha humana por parte do condutor. Tendo em vista o avanço tecnológico na área de Inteligência Artificial vemos que vem se tornando viável o desenvolvimento de um modelo que seja capaz de conduzir um veículo automotivo em ambientes urbanos, dessa forma poderíamos ter em vias públicas condutores que não se distraiam e não cometam infrações de trânsitos.

Objetivos

Objetivos Gerais

O propósito deste projeto é entregar um ambiente simulado que sirva de treinamento para veículos autônomos, que possa ser útil não somente para os estudos e análises que serão realizados neste mas também em futuros trabalhos que envolva treinar agentes em um cenário urbano. O objetivo aqui é criar o ambiente, o veículo, sua mecânica de direção e um agente que consiga conduzir o veículo. No fim, o agente deverá ser capaz de conduzir o veículo em qualquer rota traçada até o destino indicado.

Objetivos específicos

Este trabalho pretende servir de base para a criação desta plataforma. Na seção Desenvolvimento, será explicado que o agente deverá superar cada desafio, com uma dificuldade crescente após o outro. Ao fim será respondido as seguintes perguntas:

- Quais são as observações mínimas que um agente precisa ter para realizar sua tarefa?
- Quais ajustes de parâmetros necessários para ter um treino efetivo?
- Como testar o agente treinado?

Etapas do estudo

Podemos abstrair o treino em três segmentos: **ambiente**, **agente** e **algoritmo**. Ambiente engloba qualquer coisa que o agente interage, por exemplo, o cenário urbano como

as ruas e calçadas, e uma inclusão de elementos de trânsito como outros veículos, semáforos ou pedestres seriam alterações no ambiente. O segundo segmento se refere ao aprendiz, alterações feitas nas informações recebidas pelo agente como distância dos obstáculos mais próximos, velocidade do veículo, distância e localização do destino, etc, mudanças feitas nestas propriedades ou em semelhantes são alterações no agente. Finalmente, algoritmo se refere ao esquema dos algoritmos usados (PPO ou SAC) e seus hiper-parâmetros.

A primeira etapa deste estudo o agente terá de aprender a conduzir como se estivesse sozinho, isso permitirá entender melhor o que é necessário para fazer com que o aprendiz precisa para percorrer qualquer trajeto dado como objetivo. Isto é o ambiente inicialmente será apenas uma cidade vazia e a cada etapa deve-se aumentar a complexidade dele visando se aproximar da realidade. Em cada etapa as mudanças devem seguir a ordem dos segmentos, com uma certa flexibilidade, ou seja, um aumento de realismo no ambiente deve levar a alterações sensoriais no agente para se adaptar ao novo cenário e calibrar os hiper-parâmetros dos algoritmos usados.

Em cada etapa então deve-se:

- Analisar quantos e como serão dispostos os sensores o veículo deve possuir;
- Como é o desempenho usando cada algoritmo (PPO ou SAC), como as alterações dos hiper-parâmetros afetam o treino e a convergência/otimização do comportamento;

Parte I

Preparação da pesquisa

1 Fundamentação teórica

Neste capítulo será introduzido 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, 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 um 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 a frente. Os níveis dois e três 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 três o veículo assumiria o controle por um período, podendo até fazer manobras mais avançadas, como ultrapassar um automóvel mais lento a 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 que o motorista estivesse a condução, só assumiria o controle quando necessário, 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, nesse artigo 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 usam 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 pelo veículo,

isto é, o cálculo do percurso da origem ao destino, imaginar uma cidade com suas ruas, intersecções e pontos de destino como se fosse um grafo ponderado com arestas e vértices e aplicar um algoritmo de busca como Djikstra não é o suficiente para gerar uma solução eficiente, então para isso requer algo que se utilize de outros dados como informação histórica e em tempo real para achar o melhor caminho a ser percorrido em um dado dia da semana, em um dado horário e clima.

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 inclui 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. Com 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 (Open Source sob licença MIT), embora haja uma preferência natural por uma ferramenta open source do que um software proprietário para realizar esta tarefa, a Unity 3D possui uma comunidade muito maior que o Godot Engine de acordo com (ITCH.IO (2023)) e (SteamDB (2023)), ambas as fontes são portais que servem para divulgação e venda de jogos publicados por desenvolvedores individuais ou empresas, a Unity 3D é mais utilizada em ambas as plataformas. A Unity 3D possui também seu 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. 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,

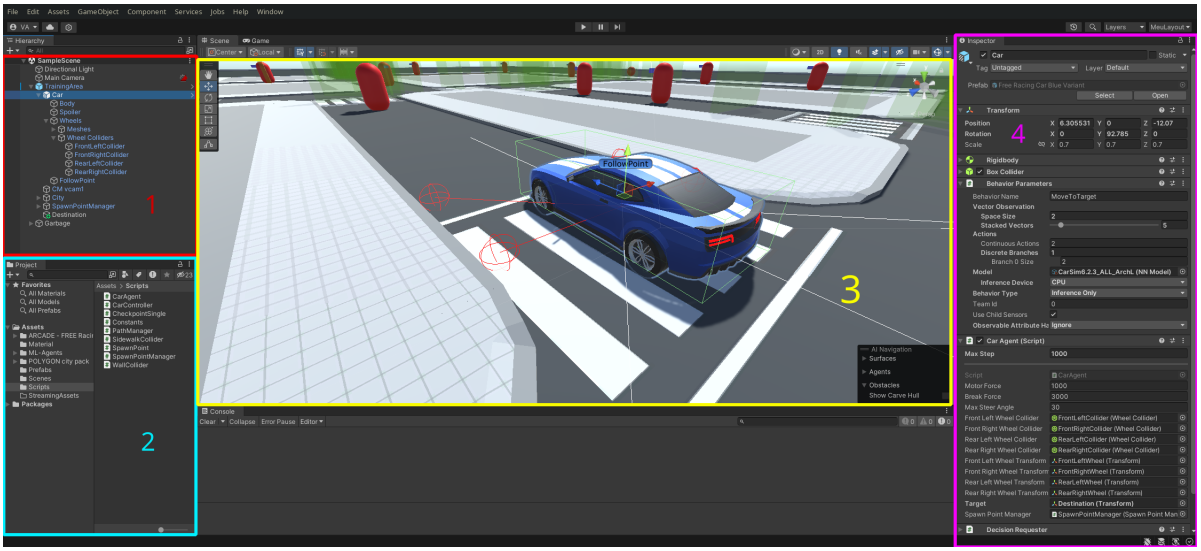

Figura 1 – Interface da Unity3D. A janela 1 é a hierarquia, 2 é a janela de projeto, 3 é visualização da cena e 4 é o inspetor

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

Os conceitos de como funciona um desenvolvimento de jogo dentro do editor da Unity3D podem não ter ficado claro ao leitor, porém na parte sobre o sistema proposto deste artigo é explicado em mais detalhes a estrutura do projeto.

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. Porém, para compreender melhor o seu escopo, pode-se dizer que está interessada 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"(Russell e Norvig (2009)).

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 artigo 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 uma 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.

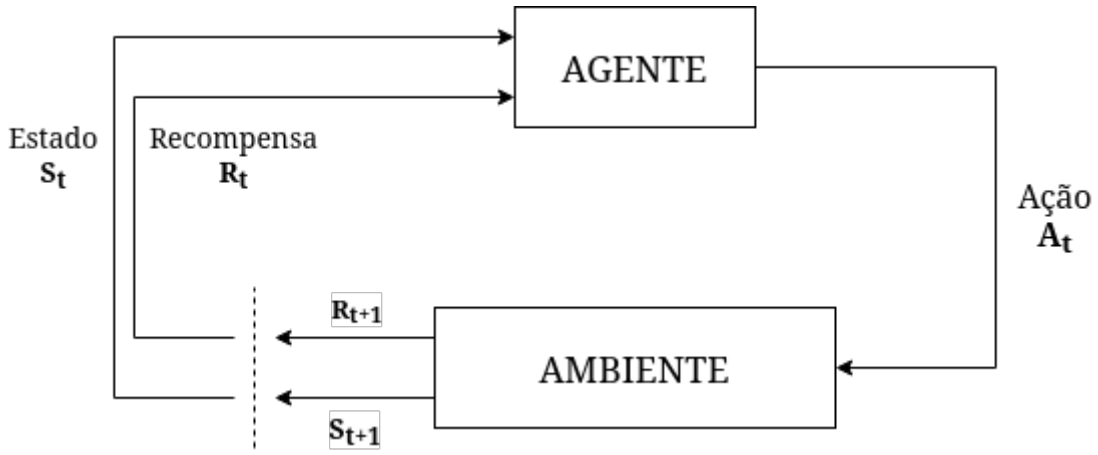

Figura 2 – diagrama da interação do agente com o ambiente. Adaptado de Sutton e Barto (2018)

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 define um vencedor ou define um empate onde nenhuma jogada mais pode ser feita, 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 $0 \leqslant \gamma \leqslant 1$:

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

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.

Com isso chegamos a uma das partes centrais do RL, 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 funcão 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.

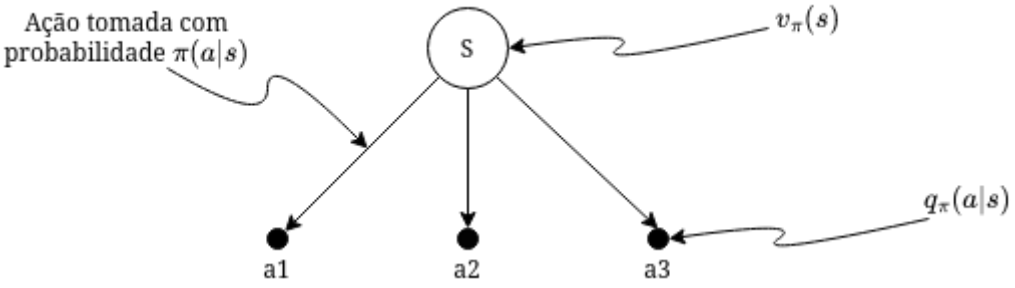

Figura 3 – Diagrama das funções valores. Ao estado s é atribuído o valor $v_\pi(s)$, a cada uma das ações é atribuído o valor $q_\pi(a_n|s)$ e tem probabilidade $\pi(a_n|s)$ de ser selecionada. Adaptado de Sutton e Barto (2018)

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 com 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 funcões 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 disturbio 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 aprendizado por no início do treino suas ações serem randômicas existindo uma enorme

dificuldade em fazer um agente aprender uma tarefa executada por humanos. Para isso existe 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.

2 Estado da Arte

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

TORCS e CARLA

TORCS é um simulador *open-source* bastante modular que pode ser usado para a criação de agentes artificialmente inteligentes. Sua portabilidade e modularidade é ideal para que se possa modificar o ambiente a fim de aumentar os desafios dos agentes que serão treinado 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 artigo 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 o 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 leva em conta 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.

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 Untiy3D que adiciona os componentes e classes de **RL IL**, esta está na versão **2.0.1**. O outro pacote seria o pacote Python, 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 depêndencias Python estão presentes no requirements.txt disponíveis online no repositório oficial deste projeto que se encontra em https://github.com/antunesvitor/SimuladorDeConducao

Tanto a Unity quanto 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 servirá 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 é 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 em 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. Porém outros elementos estáticos comuns de uma cidade como calçadas, postes, árvores, prédios, etc. Portanto, o foco poderá se manter no estudo de fazer um agente percorrer as rotas. Às bordas do cenário há muros que limitam o alcance do veículo, foi concluído pelo autor que o tamanho atual é suficiente para os treinos iniciais, em estudos futuros pode ser considerado a expansão do cenário. O agente recebe uma punição ao se chocar com a calçada ou com os muros da borda da cidade.

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. Há 17 percursos predefinidos, cada uma delas 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. Assim como o tamanho do cenário, novos trajetos podem ser considerados em estudos futuros.

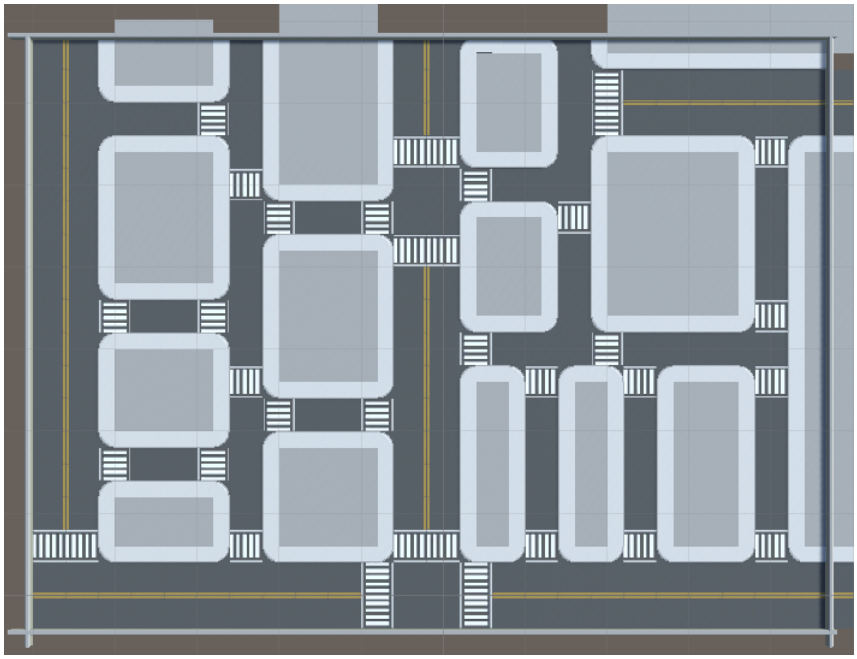

Figura 4 – Visão superior o cenário urbano criado para o treinamento do veículo

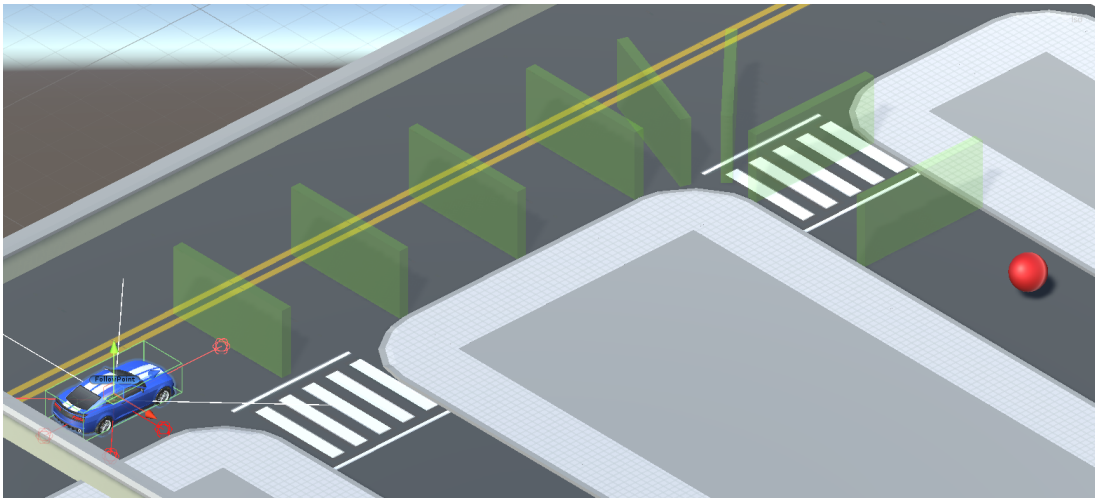

Figura 5 – Rota vista de perspectiva isométrica, o agente posicionado a origem ao canto esquerdo, com os *checkpoints* ao longo do percurso até o destino no canto direito.

4.1.2 O agente

Para o aprendizado o veículo possui 8 sensores apontando para todas as direções uniformemente espaçadas, estes sensores são um *component* do pacote **ML-agents** da Unity3D, são capazes de medir a distância dos objetos próximos ao veículo e também são capazes de distinguir quais são estes objetos. 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.

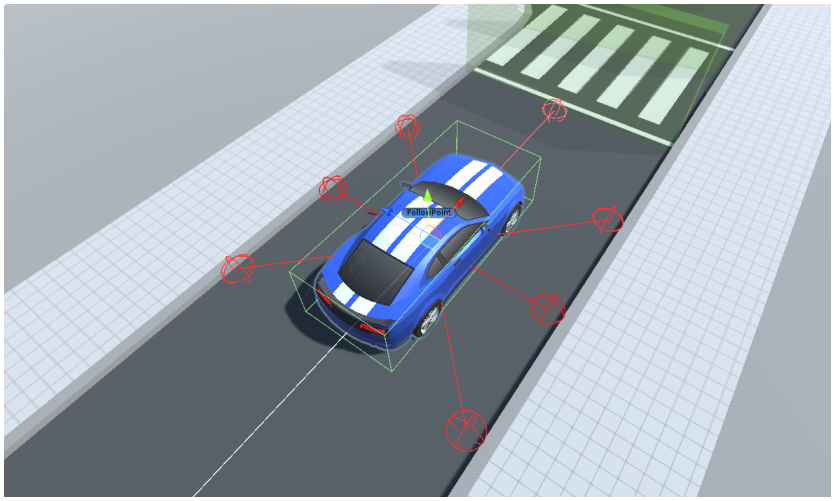

Figura 6 – Exemplo do agente e seus raios perceptores, é possível ver 6 raios laterais se chocando com as calçadas ao lado, o raio frontal se chocando com o *checkpoint*.

Além dos sensores, foi adicionado a velocidade do veículo a 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

Os eventos sujeitos a aplicação de recompensa/punição são os seguintes: o veículo chegar ao destino (R_d), o veículo atingir um checkpoint (r_c), colisão com a calçada (p_c), colisão com a parede(p_p), tombar o veículo (P_t) e por fim a punição total por *step* P_{step}. R_d, e P_t são constantes e são eventos que definem o fim do episódio. Por outro lado, r_c, p_c, p_p e p_{step} serão aplicadas ao longo do episódio conforme os eventos ocorrerem, no caso do último o evento é o próprio *step* e seu valor é $p_{step} = \frac{P_{step}}{N}$ sendo N o número máximo de *steps* por episódio.

Para o encerramento do episódio temos os seguintes cenários possíveis: o agente chega ao destino, o agente não chegar a tempo, o agente tombar o veículo. As recompensas estão no intervalo $[-1, 1]$ e serão aplicadas da seguinte forma em cada cenário:

- 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: $R_T = P_t \doteq -1$

Sendo R_T a recompensa total do episódio, R_c, P_c e P_p representam a soma total de ocorrências de r_c, p_c e p_p. Vale ressaltar que $R_d + R_c = 1$, isto quer dizer que se o veículo chegar ao destino sem se chocar com a calçada ou parede, terá a recompensa máxima, por outro lado $P_t = -1$, indicando que caso o veículo tombe, não interessa quantos checkpoints ele atingiu, a punição é máxima. Podemos interpretar o segundo cenário como: caso o veículo não chegue a tempo recebe uma recompensa pelo progresso (R_c) descontado com punição por tempo mais as punições por colisões. Sobre r_c, seu valor está sujeito ao trajeto do episódio, como eles possuem tamanho diferente, o valor a recompensa segue na forma $r_{c_x} \doteq \frac{R_c}{n_{c_x}}$, sendo n_{c_x} a quantidade de checkpoints naquela rota x.

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.

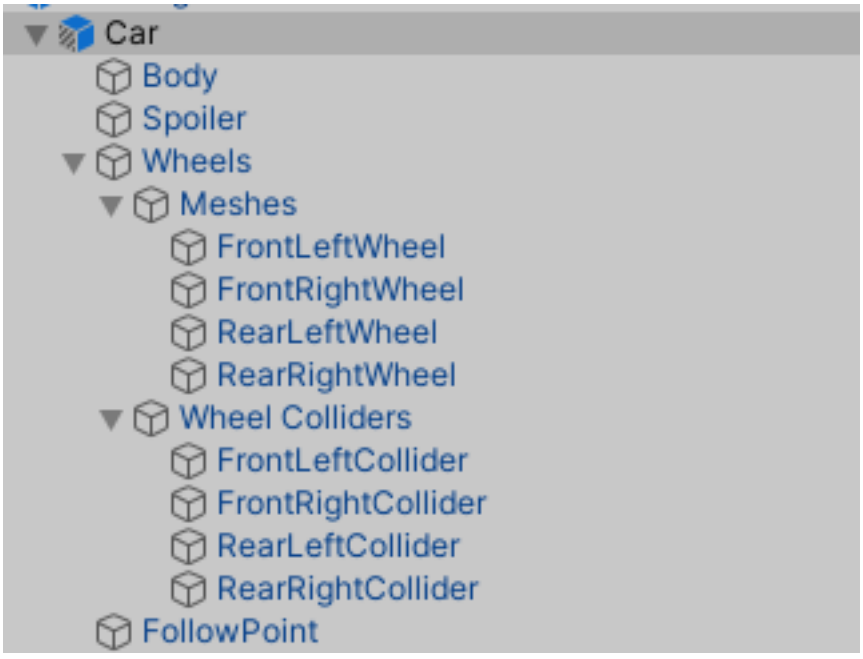

Figura 7 – Hierarquia dos *GameObjects* que compõe o veículo.

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,

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 (PrismY-outube (2022)), neste código ele 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 "horizontal"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 feito 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 hiper-parâmetros

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 hiper-parâmetros 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. Aumenta-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 corresponde a 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 ser 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

A análise e desenvolvimento do agente se dará tanto no treino quanto nos testes. 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 cometa erros. Antes que se chegue a este nível é preciso que ele demonstre sucesso em treinos mais simples, então dividimos as etapas em 4 **desafios**. Os três 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.

Como foi dito na seção Objetivos, o estudo não propõe um modelo imutável, ao invés disso, é esperado que o agente evolua em diversas frentes: seus sensores, os métodos de treino, os algoritmos, o ambiente envolvido se torne mais realista, entre outros. Como será visto no Capítulo 5, o agente e ambiente passou por mudanças a fim de cumprir as tarefas, estas alterações serão justificadas e compreendidas com os resultados dos treinos e testes.

4.3.1 Treinamento e teste

Tendo o Modelo e Os algoritmos e hiper-parâmetros detalhados acima é preciso explicar o treinamento e teste. O treinamento é executado via linha de comando, é necessário ativar o ambiente Python, gerar um executável do simulador pelo editor da unity e ter um arquivo de configuração do algoritmo. Tendo estes três itens é possível executar o comando do ml-agents e por o agente em treino. Após o treino finalizar é gerado um arquivo **.onnx** que é política final do agente, a rede neural com os coeficientes gerados pelo algoritmo após 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 como a política traduz os estados em ações, definida no arquivo .onnx, sem qualquer input do usuário, e o mesmo deve estar agindo de acordo com os resultados do treino, seja ele um sucesso ou não.

Nos testes o agente deverá percorrer os mesmos trajetos, agora eles não serão aleatórios, ele percorrerá cada um três vezes. Como na lista de observações não inclui qualquer informação sobre qual trajeto ele está percorrendo no momento, o autor não viu necessidade de separar trajetos de treino e de teste, embora essa possibilidade posta em estudos futuros.

Estatísticas de treino

Quando o treino é finalizado é armazenado diversas estatísticas que são exibidas pelo **tensorboard**. Nelas é possível ter uma visualização do desempenho do agente durante o treino e sua curva de aprendizado. As que serão analisadas e discutidas na próxima parte estão descritas na lista abaixo. Assimi como a listagem dos hiper-parâmetros essas definições são traduções diretas da documentação do ML-agents (Juliani et al. (2020)).

Cumulative Reward: A mediana da recompensa acumulada no episódio do agente. Deve aumentar ao longo de um treino bem sucedido (Esta estatística possui dois gráficos, um de linha e outro histrograma)

Episode Length: A duração mediana dos episódios (No caso deste projeto a duração deve diminuir ao longo do tempo, pois significa que o agente está realizando a tarefa mais eficientemente).

Entropy: a entropia da política, isto é, o quão aleatório as decisões do modelo são. Deve decair durante um treino bem sucedido. Se decair muito rapidamente deve-se aumentar o valor do hiperparâmetro **beta**.

Extrinsic value estimate: É a mediana do valor de todos os estados visitados pelo agente. Deve aumentar ao longo de um treinamento bem sucedido.

Os demais gráficos são o **beta**, **epsilon** e **learning_rate** eles são basicamente o valor destes hiper-parâmetros ao longo do treino, dependendo se os *schedules* deles forem constante ou linear, é esperado uma linha constante no primeiro caso ou uma queda do valor deles até zero caso linear.

4.3.2 Estudo e análise

Para cada desafio será avaliado o treino e teste. A análise do treino envolverá(além dos gráficos mencionados acima) o tempo que demorou para convergir, a mediana e desvio padrão da recompena final. Para o teste, teremos duas formas diferentes de avaliar. Para os três primeiros desafios, o agente deverá percorrer o percurso que treinou dez vezes, será avaliado pela recompensa média e a quantidade de vezes que chegou ao destino. Para o desafio geral, deverá percorrer todas as rotas três vezes, será avaliado da mesma forma, por rota e o consolidado total. O primeiro desafio será específico em uma rota retilínea, sem necessidade de fazer curvas. O segundo exigirá que o agente faça uma curva, será treinado em dois trajetos diferentes. O último antes do geral, exigirá mais de uma curva, será treinado em três trajetos distintos. Por fim o geral será treinado em todos os trajetos, de diferentes tamanhos e complexidade. Fora a análise quantitativa, haverá também uma análise mais subjetiva da condução do veículo.

Tabela 1 – Avaliação dos desafios. A coluna de rotas informa qual o *GameObject* de rota que será treinado. A figura deles estará nos resultados.

Desafio	Rota(s)	Qtde. testes
Específico: de trajeto fácil	Path(2)	10
Específico: de trajeto mediano	Path(8) Path(0)	10 em cada rota
Específico: de trajeto difícil	Path(3) Path(5) Path(6)	10 em cada rota
Geral	Todos	30 (3 em cada)

Aqui fica mais claro ao leitor o porquê o agente não é imutável, a cada desafio aumenta bastante a dificuldade da tarefa, exigindo uma calibragem mais sensível de hiper-parâmetros ou um sistema sensorial mais robusto. Neste capítulo foi ilustrado a versão mais básica do veículo e do ambiente mas no próximo, onde veremos sobre os resultados dos treinos e testes, será expostos as mudanças que precisaram ser feitas no modelo.

No teste de um desafio, o agente deverá completar ao menos 90% dos trajetos e média de recompensa acima de 0,95 cumprir os requisitos antes de tentar o próximo.

Parte III

Parte Final

5 Resultados e Discussão

5.1 Desafio em trajeto especifíco fácil

O primeiro e único treino foi executado como agente percorrendo somente um percurso em linha reta, conforme a imagem abaixo.

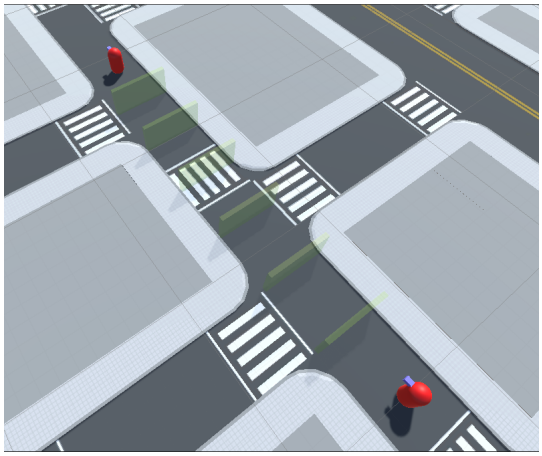

Figura 8 – O percurso executado no primeiro treino. A cápsula vermelha no canto inferior direito indica a posição inicial do veículo e a na parte superior esquerdo indica o destino.

Figura 9 – Estatísticas referentes a política. O primeiro gráfico é a entropia, decaindo como é esperado. O terceiro gráfico é o Extrinsic Value estimate, sobe e estabiliza, conforme esperado

Análise

O treino foi bem sucedido, o agente soube conduzir bem o veículo. Vendo o gráfico 1 da figura 9, pode-se notar o crescimento exponencial da recompensa e então sua estabilização quando atinge o máximo da recompensa que é possível receber. Também percebe-se que na mesma velocidade mas desta vez em sentido descendente a duração média do episódio, rapidamente cai pois o agente já dominou o trajeto. Isso é o suficiente

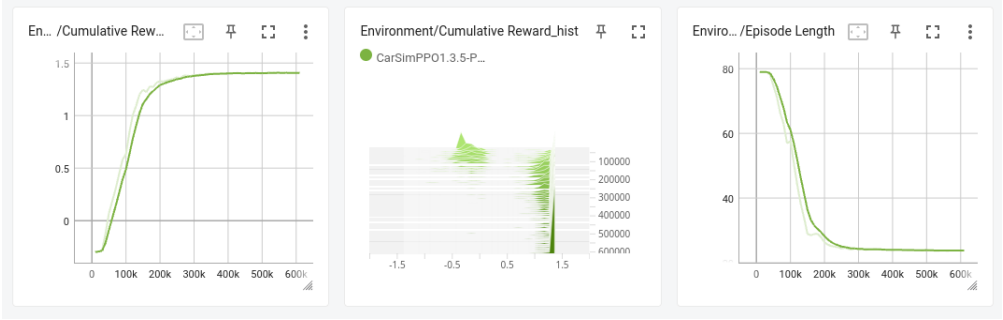

Figura 10 – Estatísticas do ambiente. O primeiro gráfico é o acumulado de recompensa, o segundo é um histograma das distribuições de recompens. O terceiro é a duração do episódio.

para este trajeto, podemos treinar o agente em um novo desafio especifíco mas com um trajeto mais complexo.

5.2 Desafio em trajeto específico dificuldade média

Aqui o treino foi feito em dois trajetos (identificados por Path(8) e Path(0) no projeto), ambos exigem que o agente faça uma conversão do veículo.

Neste desafio foi necessário uma melhoria no agente, foi aumentado o número de sensores, agora ele possui dois *components* Ray Perception Sensor 3D, o primeiro é para visão frontal e detecta os checkpoints e obstáculos e o segundo é uma visão para todas as direções do carro servindo para detecção apenas de obstáculos. Isso foi necessário pois na primeira versão havia pontos cegos, durante os testes o veículo não completava pois os sensores não detectavam o destino em sua frente.

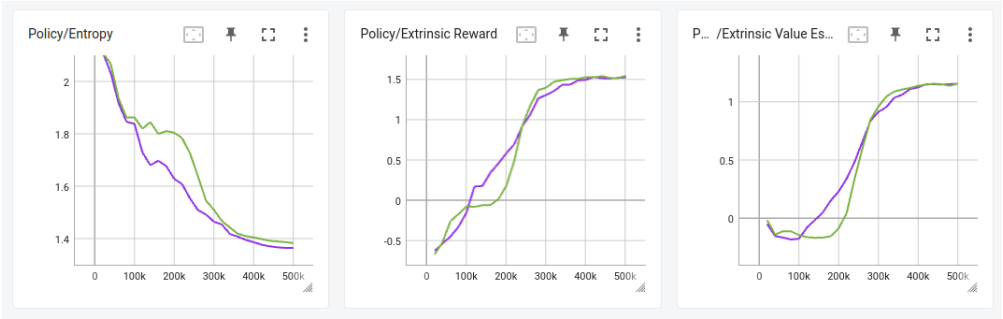

Figura 11 – Estatísticas referentes a política do segundo desafio nas duas rotas, em verde a rota *Path(8)* e em roxo a rota *Path(0)*.

Análise

O treinamento não encontrou problemas para convergir neste desafio, para ambas as rotas foram dados 500 mil passos de treino mas por volta de 400 mil passos a recompensa mediana se estabilizou no topo mostrando que o modelo convergiu. De acordo com os

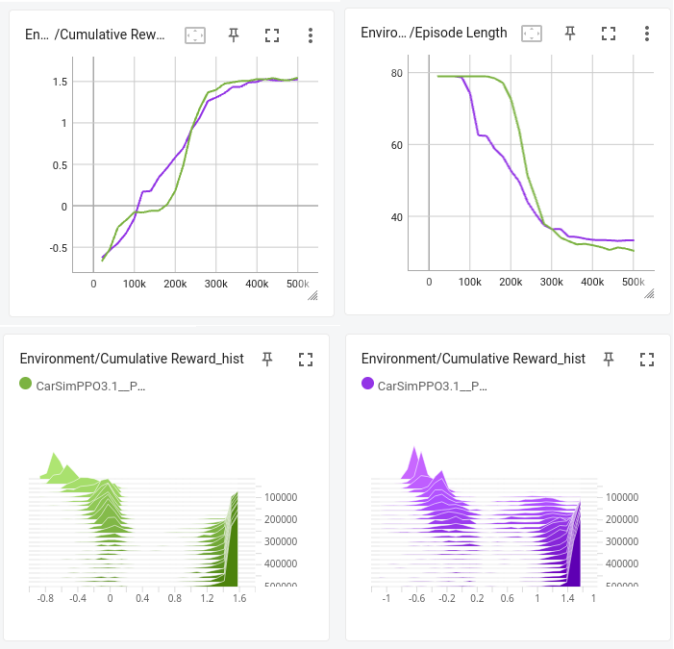

Figura 12 – Estatísticas do ambiente do segundo desafio para ambas as rotas. Em cima os gráficos de recompensa e duração do episódio, abaixo os histogramas.

histrogramas o agente conclui praticamente todos os episódios e isso se reflete nos testes onde o agente chega ao destino em todos as tentativas, subindo a calçada somente uma única vez.

5.3 Desafio em trajeto específico difícil

Para este desafio foi testado em três rotas diferentes, duas delas possui duas curvas e a terceira possui três curvas.

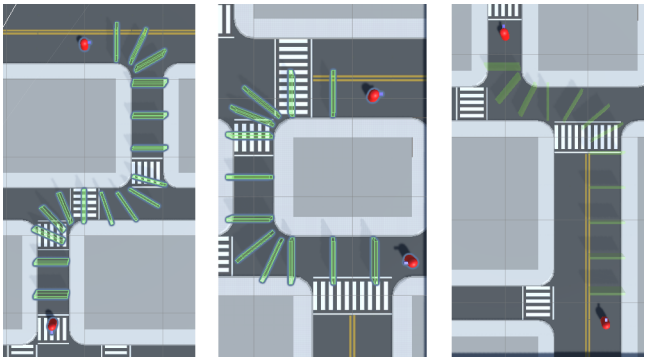

Figura 13 – As três rotas difíceis envolve mais de uma curva, a esquerda o path(3) que possui três curvas sendo o mais complexo dos trajetos. Os outros dois são o path(5) e path(6) que possuem 2 curvas cada.

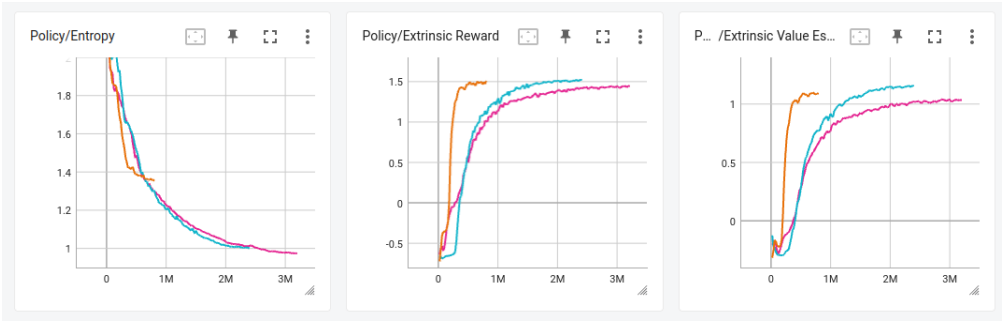

Figura 14 – Estatísticas referentes a política do desafio em rotas difíceis. *Path(3)*, *Path(5)* e *Path(6)* em laranja, azul e magenta, respectivamente.

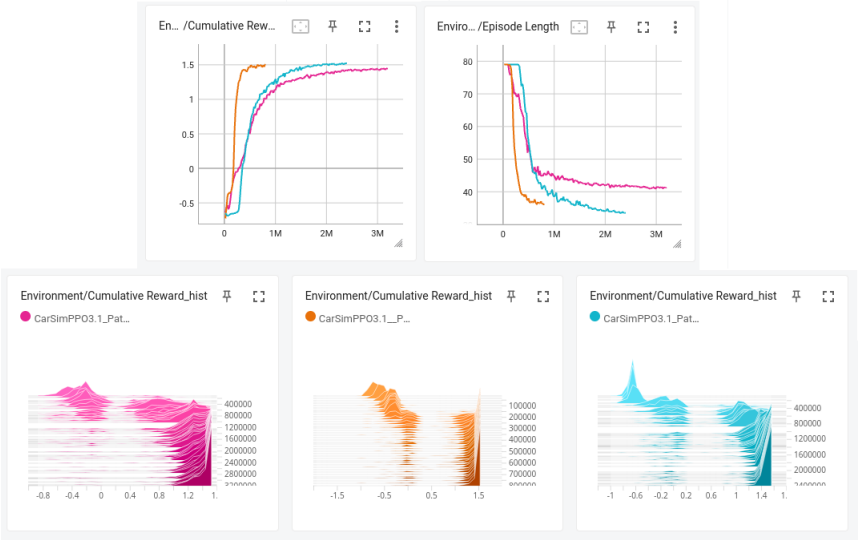

Figura 15 – Estatísticas referentes ao ambiente do desafio em rotas difíceis. Em cima os gráficos da recompensa e duração dos episódios, abaixo os histrogramas de recompensa.

Análise

Neste desafio ficou claro que o agente teve muito mais dificuldade de convergir para ter um desempenho ótimo. Embora os três trajetos estejam no mesmo grupo de desafio cada uma das três rotas exigiu uma quantidade de steps maior para covergir, o *Path(5)*, que envolvia fazer duas curvas a direita, foi o mais fácil para o agente, o treino durou 800 mil passos mas por volta de 500 mil steps já havia atingido a média máxima de recompensa. Porém, as rotas Path(3) e Path(6) demoraram respectivamente 3,2 milhões e 2,4 milhões de steps para atingir o mesmo nível, sendo que ao final do treino o primeiro ainda ficou com uma recompensa média ligeiramente abaixo dos demais.

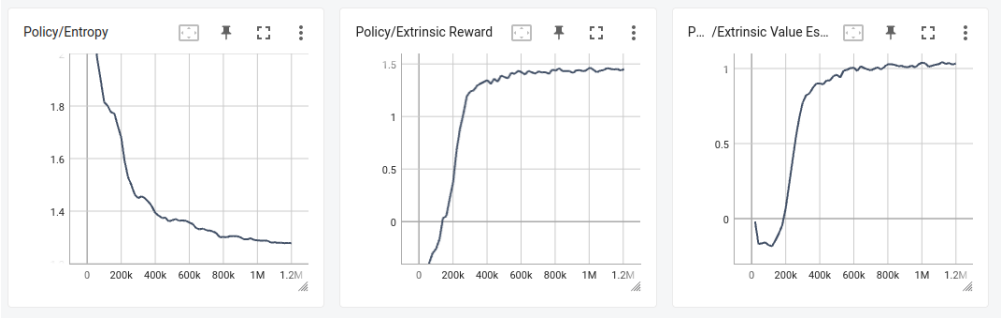

Figura 16 – Estatísticas referentes a política do desafio geral.

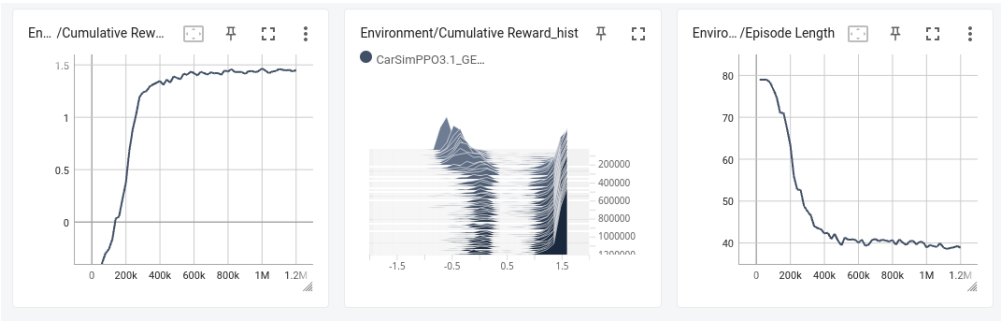

Figura 17 – Estatísticas do ambiente do desafio geral.

5.4 Desafio em todos os trajetos

Análise

O desafio foi um sucesso, o treino utilizando de PPO convergiu por volta dos 600 mil passos, metade do limite, o agente "percebeu"rapidamente que seu objetivo era atingir cada checkpoint até o destino. O teste no geral foi positivo sendo perfeito em 10 dos 17 trajetos, obtendo 0,89 de média geral e concluindo 48 dos 51 tentativas. Porém se fizermos um recorte de trajetos difíceis (duas ou mais curvas) teremos um resultado não tão satisfatório. Neste caso seriam além dos envolvidos no desafio anterior, seriam os Path's de número 7, 10, 12 e 14, teríamos 18 conclusões de 21 corridas feitas (85%), todas as vezes que o agente falhou foram em um desses e com média de apenas 0,63 de recompensa.

Tabela 2 – Consolidado do teste geral, inclui todas tentativas e suas recompensas e a recompensa média por rota.

Rota	C. T. 1	Rec. 1	C. T. 2	Rec. 2	C. T. 3	Rec. 3	Rec. média
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

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 terem sido surpreendemente rápidos (o desafio geral convergiu por volta 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.

Trabalhos Futuros

Acerca de trabalhos futuros pode-se implementar melhorias no modelo, como este projeto é uma apresentação da plataforma existe muitos elementos de trânsito que foram deixados de lado como outros veículos, pedestres, semáforos, etc.

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 8.

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 19.

HAARNOJA, T. et al. *Soft Actor-Critic Algorithms and Applications*. 2018. Citado na página 16.

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 17.

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 17.

ITCH.IO. *Most used Engines*. [S.l.], 2023. Disponível em: <https://itch.io/game-development/engines/most-projects>. Citado na página 9.

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 30 e 33.

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 11.

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 19.

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 7.

PRISMYOUTUBE. *PrismYoutube/Unity-car-controller*. 2022. Disponível em: <https://github.com/PrismYoutube/Unity-Car-Controller>. Citado na página 29.

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 17.

RUSSELL, S.; NORVIG, P. *Artificial intelligence*. 3. ed. Upper Saddle River, NJ: Pearson, 2009. Citado 2 vezes nas páginas 10 e 11.

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 7.

SCHULMAN, J. et al. *High-Dimensional Continuous Control Using Generalized Advantage Estimation*. 2015. Citado na página 31.

SCHULMAN, J. et al. *Proximal policy optimization algorithms*. 2017. Disponível em: <https://arxiv.org/abs/1707.06347>. Citado na página 16.

STEAMDB. *What are games built with and what technologies do they use?* [S.l.], 2023. Disponível em: <https://steamdb.info/tech/>. Citado na página 9.

SUTTON, R. S.; BARTO, A. G. *Reinforcement Learning*. 2. ed. [S.l.]: MIT Press, 2018. Citado 3 vezes nas páginas 11, 12 e 14.

TECHNOLOGIES, U. *Wheel collider component reference*. 2023. Disponível em: <https://docs.unity3d.com/Manual/class-WheelCollider.html>. Citado na página 28.

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 23.

UNITY ASSET STORE. *CITY package*. [S.l.], 2023. Disponível em: <https://assetstore.unity.com/packages/3d/environments/urban/city-package-107224>. Citado na página 23.

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 19.

WYMANN, B. et al. *TORCS, The Open Racing Car Simulator, v1.3.5*. 2013. <http://www.torcs.org>. Citado na página 19.

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.

É possível baixar e reproduzir os testes feitos neste projeto, o desempenho pode não ser exatamente igual aos dos vídeos acima, porém é esperado um resultado semelhante.

Como reproduzir os testes está descrito na página inicial do repositório, porém a fim de deixar este artigo 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 que foram feitos no projeto. Eles obterão o exato resultado mostrado no projeto dado a natureza estocástica do agente e grande quantidade 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.
- Arch Linux: instale o pacote na AUR. Para isso terá que usar um gerenciador de pacotes da AUR como o yay ou o paru.
- Para as distribuições Linux oficialmente suportadas siga as instruções na documentação.

2. Após isso clone ou baixe o repositório do projeto.
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 existe duas opções:
   - (Recomendado) Baixar a versão exata do projeto: Vá na página de arquivo da editores da Unity 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 pra identificar o algoritmo que 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 a 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 do 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.

APÊNDICE B – Segundo apêndice com título tão grande quanto se queira porque ele já faz a quebra de linha da coisa toda

Maecenas dui. Aliquam volutpat auctor lorem. Cras placerat est vitae lectus. Curabitur massa lectus, rutrum euismod, dignissim ut, dapibus a, odio. Ut eros erat, vulputate ut, interdum non, porta eu, erat. Cras fermentum, felis in porta congue, velit leo facilisis odio, vitae consectetuer lorem quam vitae orci. Sed ultrices, pede eu placerat auctor, ante ligula rutrum tellus, vel posuere nibh lacus nec nibh. Maecenas laoreet dolor at enim. Donec molestie dolor nec metus. Vestibulum libero. Sed quis erat. Sed tristique. Duis pede leo, fermentum quis, consectetuer eget, vulputate sit amet, erat.

Donec vitae velit. Suspendisse porta fermentum mauris. Ut vel nunc non mauris pharetra varius. Duis consequat libero quis urna. Maecenas at ante. Vivamus varius, wisi sed egestas tristique, odio wisi luctus nulla, lobortis dictum dolor ligula in lacus. Vivamus aliquam, urna sed interdum porttitor, metus orci interdum odio, sit amet euismod lectus felis et leo. Praesent ac wisi. Nam suscipit vestibulum sem. Praesent eu ipsum vitae pede cursus venenatis. Duis sed odio. Vestibulum eleifend. Nulla ut massa. Proin rutrum mattis sapien. Curabitur dictum gravida ante.

Phasellus placerat vulputate quam. Maecenas at tellus. Pellentesque neque diam, dignissim ac, venenatis vitae, consequat ut, lacus. Nam nibh. Vestibulum fringilla arcu mollis arcu. Sed et turpis. Donec sem tellus, volutpat et, varius eu, commodo sed, lectus. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Quisque enim arcu, suscipit nec, tempus at, imperdiet vel, metus. Morbi volutpat purus at erat. Donec dignissim, sem id semper tempus, nibh massa eleifend turpis, sed pellentesque wisi purus sed libero. Nullam lobortis tortor vel risus. Pellentesque consequat nulla eu tellus. Donec velit. Aliquam fermentum, wisi ac rhoncus iaculis, tellus nunc malesuada orci, quis volutpat dui magna id mi. Nunc vel ante. Duis vitae lacus. Cras nec ipsum.

Anexos

ANEXO A – Nome do Primeiro Anexo

Sed mattis, erat sit amet gravida malesuada, elit augue egestas diam, tempus scelerisque nunc nisl vitae libero. Sed consequat feugiat massa. Nunc porta, eros in eleifend varius, erat leo rutrum dui, non convallis lectus orci ut nibh. Sed lorem massa, nonummy quis, egestas id, condimentum at, nisl. Maecenas at nibh. Aliquam et augue at nunc pellentesque ullamcorper. Duis nisl nibh, laoreet suscipit, convallis ut, rutrum id, enim. Phasellus odio. Nulla nulla elit, molestie non, scelerisque at, vestibulum eu, nulla. Ut odio nisl, facilisis id, mollis et, scelerisque nec, enim. Aenean sem leo, pellentesque sit amet, scelerisque sit amet, vehicula pellentesque, sapien.

ANEXO B – Nome de Outro Anexo

Phasellus id magna. Duis malesuada interdum arcu. Integer metus. Morbi pulvinar pellentesque mi. Suspendisse sed est eu magna molestie egestas. Quisque mi lorem, pulvinar eget, egestas quis, luctus at, ante. Proin auctor vehicula purus. Fusce ac nisl aliquam ante hendrerit pellentesque. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Morbi wisi. Etiam arcu mauris, facilisis sed, eleifend non, nonummy ut, pede. Cras ut lacus tempor metus mollis placerat. Vivamus eu tortor vel metus interdum malesuada.