Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Jogos digitais como ferramenta eficiente de coleta de dados para IAs de vigilância

Thiago Santos Teixeira

Monografia Final

mac 499 — Trabalho de Formatura Supervisionado

Supervisor: Prof. Flavio Soares Correa da Silva

São Paulo
2022

”Dedico este trabalho a minha família,
cujo amor e apoio foram fundamentais.”

Agradecimentos

To get what you love, you must first be patient with what you hate.
— Al-Ghazali

Aos meus familiares, por me apoiarem em qualquer circunstância, à minha namorada por ser minha companheira constante me motivar a chegar ao final, aos meus amigos por serem fonte constante de apoio e tornaram a jornada mais suportável e aos meus colegas de trabalho que me guiaram quando precisei.

Resumo

Thiago Santos Teixeira. **Jogos digitais como ferramenta eficiente de coleta de dados para IAs de vigilância**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2022.

Este trabalho busca desenvolver um jogo digital que sirva como ambiente de treinamento para IAs de vigilância que possam ser utilizadas em situações reais. Para atingir esse objetivo foram utilizados algoritmos de aprendizado por reforço, nos quais um agente aprende baseado em interações com o ambiente e busca ações que maximizem a possível recompensa ao longo do tempo. Para atingir o produto final, foram necessárias diversas interações que foram feitas utilizando outro agente para que o jogo se tornasse minimamente interessante ao usuário. Por fim, o projeto possui recursos para ser expandido até um jogo completo.

Palavras-chave: Jogos digitais. Aprendizado por reforço. Vigilância.

Abstract

Thiago Santos Teixeira. **tcc**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2022.

This work seeks to develop a digital game that serves as a training environment for surveillance AIs that can be used in real situations. To achieve this goal, reinforcement learning algorithms were used, in which an agent learns based on interactions with the environment and seeks actions that maximize the possible reward over time. To reach the final product, it took several interactions that were made using another agent so that the game became minimally interesting to the user. Finally, the project has the resources to be expanded into a full game.

Keywords: Digital Games. Reinforcement Learning. Surveillance.

Lista de Figuras

Lista de Tabelas

Lista de Programas

Sumário

Anexos

Capítulo 1

Introdução

1.1 Motivação

A segurança é uma preocupação crescente em todo o mundo, e garantí-la em edifícios e instalações é fundamental para a tranquilidade e bem-estar das pessoas. Por isso, é muito comum, especialmente no Brasil, que sejam contratados vigias. O cargo de vigia é tradicionalmente entendido como aquele que tem como função principal a vigilância e segurança de edifícios, instalações e propriedades. Entretanto, com o avanço da tecnologia e a crescente necessidade de segurança eletrônica, é possível questionar a relevância e a eficácia desse cargo no atual cenário.

Além do fato de que em muitos casos, esses profissionais são colocados em situações de risco, como a necessidade de lidar com violência e conflito, que podem levar a problemas de saúde mental e física (PATEL; SINGH, 2019). Um estudo publicado no "International Journal of Occupational Safety and Ergonomics"(TÜRKER et al., 2016) descobriu que os vigilantes são mais propensos a sofrerem problemas psicológicos, como estresse, ansiedade e depressão, devido à natureza isolada e monótona do trabalho. Uma publicação na "Revista Brasileira de Saúde Ocupacional"(GOMES et al., 2011) aponta que também são propensos a sofrer de problemas de saúde física, como dores nas costas, devido à natureza sedentária do trabalho.

Por isso, empresas e centros de estudo estão desenvolvendo robôs automatizado com inteligência artificial e machine learning para realizar o papel de vigia. Um deles é o robô Knightscope K5 (KNIGHTSCOPE, 2023), que é equipado com câmeras de alta definição, reconhecimento facial, reconhecimento de placas de veículos, reconhecimento de som e sensores de temperatura e umidade, ele pode detectar atividades suspeitas, como aglomerações de pessoas, objetos abandonados e veículos estacionados de forma inadequada. Outras capacidades são se comunicar com as autoridades em caso de emergência e realizar tarefas casuais, como verificar os bilhetes de entrada, verificar a documentação de veículos, monitorar o tráfego e etc.

Em resumo, o cargo de vigia é obsoleto e apresenta riscos desnecessários ao ser humano. A tecnologia tem se mostrado uma alternativa mais eficiente e segura para garantir a segurança de edifícios e instalações e é importante que as empresas e instituições reavaliem

Figura 1.1: *Knightscope K5 robot*

suas necessidades de segurança e considerem a implementação de sistemas de segurança eletrônicos, bem como a capacitação dos vigias para operar esses sistemas.

1.2 Objetivo

Este trabalho busca desenvolver um jogo digital que sirva como ferramenta de coleta de dados eficiente para o treinamento de IAs de vigilância. Ou seja, deve apresentar um mapa similar a um ambiente real, e através das decisões do jogador, o modelo deve tentar atingir uma patrulha ótima para o mapa, buscando proteger o máximo possível de objetivos e impedir que o protagonista fuja.

O jogo deve coletar os dados de cada rodada de maneira eficiente, e se adaptar aos padrões e estratégias identificados. Para isso, parte do processo também será se manter divertido para o usuário, e manter uma alta re-jogabilidade, ou seja, que ele possa ser jogado várias vezes e portando colete a maior quantidade de dados possível.

Ao final do experimento, esperamos ter uma IA que seja capaz de patrulhar com sucesso o ambiente em que trabalha usando apenas os dados que foram coletados de rodadas passadas, e que possa ser usada em um ambiente real de modo a garantir maior segurança e monitoramento.

1.3 Organização do Texto

O texto está organizado em outros 4 capítulos: o capítulo 2 apresenta os conceitos e ferramentas utilizados no projeto, separados em 5 subsessões. O capítulo 3 introduz o funcionamento do jogo criado e as motivações para sua criação. O capítulo 4 introduz os passos necessários para treinar o agente do inimigo do jogo, incluindo sessões separadas para as ações do agente, as observações que ele faz do ambiente e os hiper-parametros do modelo. E por último um capítulo de conclusão com os dados coletados do modelo e a conclusão do projeto.

Capítulo 2

Conceitos e ferramentas

2.1 Aprendizado de máquina

O *Machine Learning* (Aprendizado de Máquina, em tradução livre) é uma subárea da inteligência artificial que se baseia na ideia de que é possível fazer com que as máquinas aprendam por si próprias, isto é, sem a necessidade de programação explícita. Os algoritmos desse tipo são alimentados com dados e treinados de forma para que possam encontrar padrões e fazer previsões.

Atualmente, há três categorias principais de algoritmos de *machine learning*: aprendizado supervisionado, aprendizado não-supervisionado e aprendizado por reforço.

No primeiro, as máquinas são alimentadas com dados rotulados, ou seja, dados que já foram classificados previamente. A partir daí, o algoritmo de aprendizado é capaz de fazer previsões precisas para novos dados, desde que eles sejam semelhantes aos dados já vistos. Por exemplo, um estudo usou aprendizado supervisionado para desenvolver um modelo que classificasse imagens de raios-x como normais ou anormais, com base em exemplos previamente rotulados (RAJPURKAR et al., 2017). O modelo foi usado para identificação de pneumonia e foi capaz de alcançar uma precisão de aproximadamente 90% na classificação de imagens de pacientes.

Já no aprendizado não supervisionado, as máquinas são alimentadas com dados não rotulados. O objetivo é encontrar padrões nos dados, como clusters ou associações, sem a necessidade de uma classificação previa. Por exemplo, um modelo pode ser usado para identificar padrões e relações em grandes conjuntos de dados financeiros, como dados históricos de preços de ações, como o feito em (BISOI; DASH; PATRA, 2017).

No aprendizado por reforço, as máquinas aprendem a partir de uma série de ações e consequências. O objetivo é encontrar a ação que resulta na maior recompensa. Por exemplo, um modelo de aprendizado por reforço pode ser usado para controlar agentes a não derrubar uma bola em um ambiente virtual, como o caso apresentado em (TECHNOLOGIES, 2021a).

Devido à sua natureza de aprendizado a partir de dados, o aprendizado de máquina é útil em uma ampla variedade de aplicações, incluindo reconhecimento de imagem,

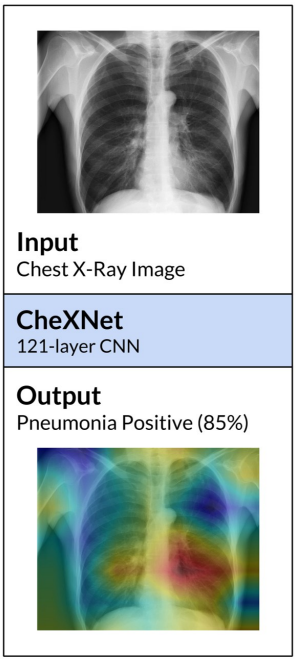

Figura 2.1: *Resultado apresentado pelo modelo de aprendizado supervisionado, após análise de paciente com pneumonia*

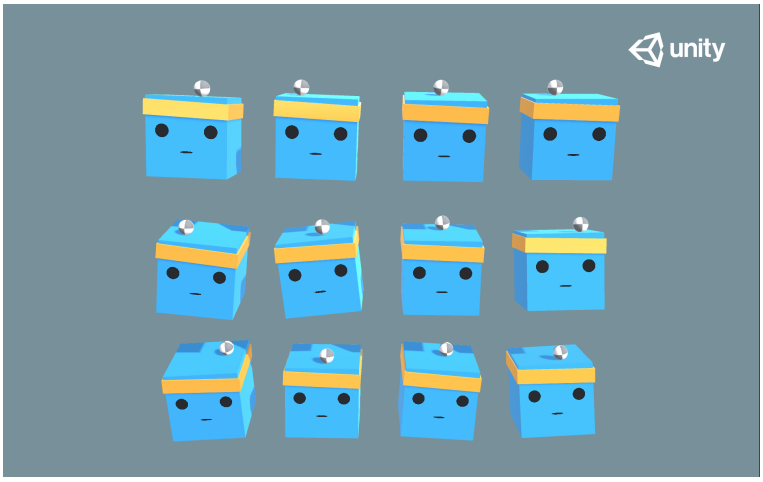

Figura 2.2: *Exemplo do uso de aprendizado por reforço apresentado na biblioteca ML-Agents*

processamento de linguagem natural, análise de dados financeiros e muito mais. É uma tecnologia poderosa e em constante evolução, e seu uso está se tornando cada vez mais comum em muitos setores.

2.2 Unity Engine

Um engine de jogos é um *software* que fornece a infraestrutura básica para o desenvolvimento e criação de jogos eletrônicos. Ele é responsável por gerenciar aspectos técnicos do jogo, como a renderização gráfica, física, som, inteligência artificial, entre outros, deixando ao desenvolvedor a tarefa de criar o enredo, personagens, regras e mecânicas de jogo.

De acordo com Oliveira (OLIVEIRA, 2010), um engine de jogos é uma plataforma que permite aos desenvolvedores criarem jogos de forma mais eficiente, sem precisar recriar a roda a cada novo projeto. Além disso, ele fornece uma série de ferramentas e recursos que facilitam o trabalho, como sistemas de iluminação, animação, partículas e outros efeitos visuais, bem como sistemas de som e física.

A utilização de um engine de jogos é amplamente difundida no setor de desenvolvimento de jogos, sendo que existem diversas opções disponíveis no mercado, cada uma com suas próprias características e recursos. Alguns exemplos de engines de jogos incluem o Unity, Unreal Engine, CryEngine, entre outros (CHAVES, 2018).

A primeira mencionada, a Unity Engine, é uma plataforma de desenvolvimento de jogos e aplicativos interativos amplamente utilizada na indústria de entretenimento digital. Desenvolvida pela Unity Technologies. A engine foi lançada em 2005 e desde então tem sido utilizada por desenvolvedores de todo o mundo para criar jogos para uma ampla variedade de plataformas, incluindo PC, dispositivos móveis, consoles de jogos e realidade virtual.

A ferramenta é conhecida por sua facilidade de uso e sua flexibilidade, o que permite que desenvolvedores criem jogos em uma ampla variedade de gêneros, incluindo jogos de ação, jogos de corrida, jogos de estratégia e jogos de realidade virtual, e combinando isso com o fato de possuir uma ampla comunidade de desenvolvedores que criam e compartilham recursos, como scripts, modelos 3D e plugins, ela ajuda a tornar o processo de desenvolvimento mais fácil e eficiente (STONE, 2017).

2.3 ML-Agents

ML-Agents é uma biblioteca *open source* disponível para Unity que oferece um conjunto de ferramentas de inteligência artificial desenvolvidas pela própria *Unity Technologies* para permitir a criação de aplicações de aprendizado de máquina em jogos e simulações. Ela se integra ao motor de jogo Unity e fornece uma infraestrutura completa para treinar agentes controlados por inteligência artificial usando algoritmos de aprendizado por reforço.

A biblioteca é uma solução escalável e flexível que permite aos desenvolvedores treinar agentes tanto localmente como em nuvem, utilizando recursos de computação paralela para acelerar o processo de treinamento. Ao mesmo tempo, fornece uma ampla gama de

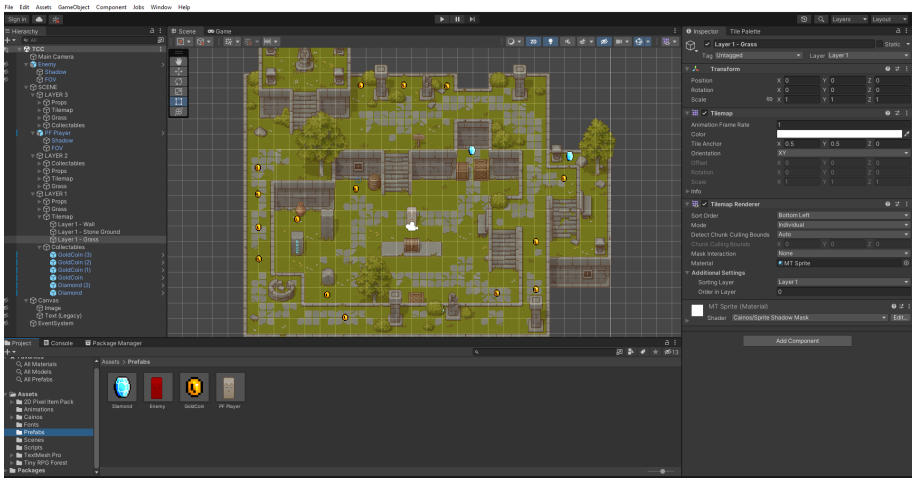

Figura 2.3: *Interface de desenvolvimento da Unity Engine*

algoritmos de aprendizado de máquina, incluindo o aprendizado por reforço, o aprendizado supervisionado e o aprendizado não-supervisionado, permitindo aos desenvolvedores escolher a abordagem mais adequada para suas necessidades (TECHNOLOGIES, 2021b).

Um ponto crucial dessa ferramenta é sua integração ao ecossistema de inteligência artificial da Unity, incluindo o *Unity Machine Learning Agents Toolkit* (TECHNOLOGIES, 2021c), que fornece uma interface visual para treinar e testar agentes, e o *Unity Inference Engine*, que permite aos desenvolvedores executar modelos de inteligência artificial treinados em dispositivos de borda, como *smartphones* e dispositivos de realidade virtual.

2.4 PyTorch

O PyTorch é uma biblioteca de aprendizado de máquina *open-source* em Python desenvolvida pela Facebook. Ele foi projetado para fornecer uma alternativa a outras bibliotecas populares, como o TensorFlow, com uma abordagem mais intuitiva e fácil de usar para desenvolver e treinar modelos de aprendizado profundo.

É composta por vários pacotes e funções que permitem aos desenvolvedores criar, treinar e avaliar modelos de *deep-learning* de forma eficiente. Ele também fornece uma plataforma fácil de usar para experimentação e prototipagem de novos modelos. Além disso, o PyTorch é altamente otimizado para aproveitar a capacidade de processamento de GPUs, o que torna o treinamento de modelos de aprendizado profundo muito mais rápido do que em plataformas menos otimizadas (CONTRIBUTORS, 2021).

A combinação do PyTorch com a biblioteca ml-agents oferece uma plataforma altamente eficiente e fácil de usar para o desenvolvimento de modelos de aprendizado por reforço. O PyTorch fornece uma infraestrutura poderosa para criar e treinar modelos, enquanto ml-agents fornece uma interface para usar esses modelos em simulações de mundos virtuais, permitindo que os desenvolvedores aproveitem a capacidade de processamento de GPUs para treinar modelos de aprendizado por reforço de forma eficiente e rápida.

2.5 Proximal Policy Optimization (PPO)

O Proximal Policy Optimization (PPO) é um algoritmo de otimização de políticas em Aprendizado por Reforço. A principal ideia por trás do PPO é equilibrar a exploração versus explotação, permitindo que o agente aprenda a partir de seus erros e ao mesmo tempo aproveite sua política atualmente conhecida para fazer escolhas ótimas.

O PPO foi proposto por John Schulman (SCHULMAN et al., 2017) e é uma variação do algoritmo de Ação Ótima de Políticas (POA). A principal diferença entre PPO e POA é que, enquanto o POA tem uma restrição rígida na mudança da política, o PPO possui uma restrição suave, conhecida como fator de clipagem, uma medida de quanto a nova política pode divergir da política antiga, sua intenção é evitar mudanças drásticas já que podem levar a instabilidade e subdesempenho no treinamento. Isso permite que a política seja atualizada mais liberamente do que o POA mas sem perder muito desempenho.

Fator de clipagem:

$$\text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon) = \max\left(\min(r_t(\theta), 1+\epsilon), 1-\epsilon\right)$$

O algoritmo PPO funciona da seguinte maneira: primeiramente, uma amostra é coletada a partir da política atual e usada para calcular uma estimativa da função objetivo. Em seguida, a política é atualizada usando otimização por gradiente para maximizar a função objetivo estimada com o fator de clipagem aplicado para garantir que a mudança na política seja limitada. Este processo é repetido várias vezes até que a política esteja otimizada.

Na otimização por gradiente, o gradiente é usado para atualizar a política de forma a maximizar a função objetivo, e é calculado usando as derivadas parciais da função objetivo em relação aos parâmetros da política. A atualização desta é feita na direção do gradiente com uma taxa de aprendizado especificada.

Otimização por gradiente:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$$

onde α é a taxa de aprendizado.

A função objetivo a ser maximizada é dada por:

$$J(\theta) = \mathbb{E}_t\left[\min\left(r_t(\theta)A_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t\right)\right]$$

onde θ é a política, $r_t(\theta)$ é a razão de probabilidade entre a política antiga e a nova, A_t é o estimador de vantagem, e ϵ é o fator de clipagem.

O PPO tem sido amplamente utilizado em muitas aplicações de Aprendizado por Reforço, como jogos, robótica e sistemas de recomendação, devido à sua eficiência computacional, capacidade de aprender políticas estáveis e pelo fato de ser altamente escalável e poder ser aplicado em ambientes paralelos e distribuídos (MNIH et al., 2016), o que o torna uma escolha popular em muitas aplicações em larga escala.

Capítulo 3

O jogo

3.1 Conceito base

Para explorar a temática do projeto, um jogo no esquema de *hide and seek* foi desenvolvido. Consiste em um jogador controlar um personagem virtual que deve escapar dos inimigos enquanto coleta moedas espalhadas pelo ambiente. Esse tipo de jogo tem sido popular desde a década de 1990, quando os primeiros jogos eletrônicos começaram a ser desenvolvidos.

A mecânica de funcionamento do jogo é simples: o jogador controla o personagem virtual e deve desviar dos inimigos para evitar ser capturado. Ao mesmo tempo, deve procurar o máximo que conseguir de moedas e diamantes espalhados pelo ambiente e caso seja encontrado, a rodada é encerrada e ele volta ao início

O jogo é baseado em inteligência artificial, o que significa que os inimigos são programados para perseguir o jogador de forma estratégica e lógica. Sua dificuldade aumenta à medida que mais rodadas são jogadas, já que o modelo de *machine-learning* que controla os inimigos se adapta às estratégias dos usuários .

Em uma eventual evolução do jogo apresentado, poderá ser utilizada a interação do jogador com o ambiente virtual para tornar a experiência mais imersiva e realista, ou seja, o jogador conseguiria interagir com objetos no ambiente, como arbustos, caixas e portas, para se esconder dos inimigos, até eventualmente poder usar itens, como fogos de artifício ou objetos para distrair os inimigos e facilitar a fuga.

Em resumo, o jogo apresentado é uma combinação de habilidade e estratégia, onde o jogador deve desviar dos inimigos e coletar moedas para progredir. A inteligência artificial dos inimigos e a interação com o ambiente virtual tornam a experiência do jogador mais desafiadora e imersiva.

3.2 O Mapa

O mapa do jogo foi desenvolvido baseado em dois objetivos: o primeiro, torná-lo similar a um ambiente real para que o modelo tenha aplicações reais, o segundo, fazer com que a

Figura 3.1: *Jogador se escondendo do vigia*

experiência do jogador seja divertida. Para torná-lo verossímil, foi utilizado um desenho similar ao utilizado em muitos shoppings hoje em dia, um espaço térreo grande central com um arco mais alto em volta, esse tipo de desenho vem sendo muito comum em shopping modernos, como por exemplo no shopping Cidade Jardim, e se aplica perfeitamente no jogo desenvolvido.

Figura 3.2: *Shopping Cidade Jardim, (JHSF, 2023)*

Para cumprir o segundo objetivo e ainda complementar o primeiro, o mapa do jogo possui diversas moedas e diamantes espalhados estrategicamente para incentivar o jogador a percorrer o mapa inteiro, esses seriam uma representação de uma loja em um eventual ambiente real, onde os diamantes representam lojas com inventário de alto valor em relação as moedas, e portanto que devem ser vigiadas com mais atenção pelos agentes. Para atingir esse efeito, coletar uma moeda recompensa o jogador em 1 ponto, enquanto coletar um

diamante da 15 pontos, e estes estão em lugares com poucas entradas e saídas, aumentando a dificuldade para o usuário.

Figura 3.3: *Mapa do jogo com as recompensas*

3.3 O protagonista

A aparência do protagonista do jogo foi inspirada na temática ao qual o contexto está inserido, afinal, estamos falando de um ambiente em que na verdade, o protagonista não é o heroi, mas sim um impostor que deve se infiltrar e roubar o tesouro sem ser percebido, pensando nisso, o protagonista ser um gato faz todo sentido com a temática já que deve ser quieto e sútil tal qual um felino.

Ele pode se movimentar pelo mapa usando os 4 eixos bidimensionais, para cima, para baixo, para a esquerda e para a direita, e para alcancar os tesouros pode subir e descer as escadas e se esconder atras de objetos espalhados pelo mapa.

Figura 3.4: *O protagonista*

Programa 3.1 Script do coletor para adicionar os pontos de cada diamante e moeda.

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using System.Runtime.CompilerServices;
4   using System.Xml;
5   using UnityEngine;
6   using UnityEngine.UI;
7   using UnityEngine.UIElements;
8
9   public class Collector : MonoBehaviour
10  {
11     private int value = 0;
12     private bool isCollectable = false;
13
14     [SerializeField] private Text valueText;
15     private void OnTriggerEnter2D(Collider2D collision)
16     {
17        if (collision.gameObject.CompareTag("Coin"))
18        {
19           value++;
20           valueText.text = "" + value;
21           isCollectable = true;
22        }
23        else if (collision.gameObject.CompareTag("Diamond"))
24        {
25           value = value + 15;
26           isCollectable = true;
27        }
28        if (isCollectable)
29        {
30           Destroy(collision.gameObject);
31           valueText.text = "" + value;
32           isCollectable = false;
33        }
34     }
35  }
```

3.4 Os inimigos

Os inimigos são a chave principal do jogo, responsáveis por trazer entretenimento ao jogador e por servirem como agentes para o algoritmo de *machine-learning*. Dado isso, a aparência deles foi baseada no próprio protagonista, sendo uma versão raivosa dele e de certa forma "oposta".

Assim como o protagonista, ele pode se locomover pelo mapa nos eixos horizontais e subir escadas para caçar o jogador, o modelo por tras de seu "pensamento"consegue enxergar objetos ao seu redor e diferenciar moedas, diamantes e o jogador.

Capítulo 4

O aprendizado

4.1 Desenvolvendo o Agente e o comportamento

O funcionamento de um agente no projeto em questão é simples de maneira geral, como a nomenclatura ja diz, seu principal papel é de executar uma ação baseado no comportamento que lhe é dado. Ele coleta observações sobre o ambiente, determina suas proprias recompensas e baseado nisso, executa ações baseado nos chamados *Behaviour Parameters*, ou seja parâmetros de comportamento, que de início são configurados em 3 grupos.

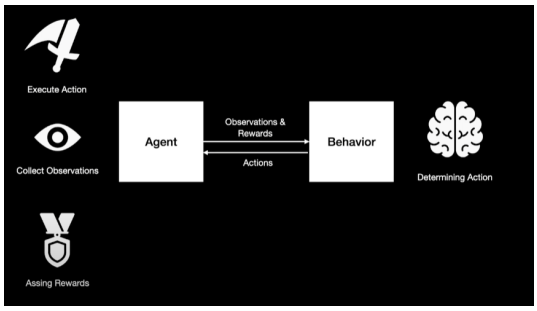

Figura 4.1: *O comportamento de um agente (OPENAI, 2021)*

4.1.1 Ações

O Primeiro é chamado de *Actions* ou ações, e nele há dois tipos diferentes de ações que o agente pode tomar: ações contínuas ou ações discretas. O primeiro tipo se refere a ações que seriam dadaos por números reais, ou *floats* na tipagem de C#, e o segundo se refere a ações que podem ser determinadas usando apenas um valor inteiro, ou seja, um *int*. Como o inimigo só pode executar ações simples, como ir para frente, trás, esquerda e direita, podemos representá-las usando apenas números inteiros, e portanto ações discretas.

Já sabendo o tipo de ação, o próximo passo foi configurar suas *Branches*. Uma *branch* representa um conjunto de ações exclusívas entre si, como por exemplo, o inimigo não pode se mover para a esquerda e para a direita simultaneamente, e portanto essas ações devem

estar concentradas em uma única *branch*. No projeto em questão, devido a simplicidade dos movimentos do jogador serão necessárias apenas duas dessas, uma para movimentos na vertical (frente e trás) e outra para os movimentos na horizontal (esquerda e direita).

Em suma, uma *branch* nada mais é que um vetor de valores que será passado para o agente onde cada elemento representa uma ação, e portanto, é necessário determinar seu tamanho. Para isso, é necessário refletir sobre quais ações o agente pode tomar em cada eixo, no eixo x (horizontal) por exemplo, o agente pode executar 3 ações diferentes: ir para a esquerda, ir para a direita e não fazer nada, portanto o tamanho de sua branch tem que ser 3, o que quer dizer que, por serem ações discretas, o agente receberá um valor entre 0 e 2 que corresponderá a ação que deve tomar para essa *branch*.

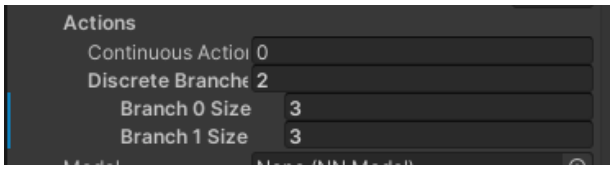

Figura 4.2: *Configuração dos parâmetros para as ações*

Com esses parametros é possível determinar o comportamento do agente ao receber cada ação. Há dois tipos de comportamento que podem ser apresentados, heurístico ou por inferência. O primeiro se refere ao que chamamos de ações *hard-coded*, nele o agente recebe os inputs de um humano e se comporta conforme mandado, bastante útil para testar o ambiente. Já o segundo se refere ao comportamento no qual as ações são determinadas pelo modelo. Esses comportamentos são tratados no código do agente abaixo:

No código, há 6 funções chaves para o comportamento do agente:

- `Initialize()`: responsável por dar ínicio ao agente, roda apenas uma vez quando o mesmo é instanciado;
- `OnActionReceived(ActionBuffers actions)`: responsável por tratar as ações vindas as *branches*, tanto ta parte heurística quanto inferida, estas são dadas como parâmetro no buffer `actions` de acordo com as configurações fornecidas anteriormente
- `Heuristic(in ActionBuffers actionsOut)`: responsável por tratar o comportamento heurístico e passa as decisões do humano para o buffer `actionsOut`, que é tratado na função anterior
- `public override void OnEpisodeBegin()`: código a ser executado no início de cada episódio.
- `private void Reset()`: código para reiniciar o ambiente ao estado inicial
- `private void OnCollisionEnter2D(Collision2D collision)`: código que trata quando o jogador se encontra com o inimigo

Programa 4.1 Classe do agente do inimigo. Parte 1

```
1
2         public override void Initialize()
3         {
4            animator = GetComponent<Animator>();
5            gameObject.layer = LayerMask.NameToLayer("Enemy");
6            startingPos = transform.localPosition;
7            startingLayer = GetComponent<SpriteRenderer>().sortingLayerName;
8         }
9         public override void OnActionReceived(ActionBuffers actions)
10        {
11           if (playerPoints < playerRef.value)
12           {
13              float changed = playerRef.value - playerPoints;
14              AddReward( - changed / 20);
15              Debug.Log(GetCumulativeReward());
16              playerPoints = playerRef.value;
17           }
18           AddReward(-0.0005f);
19           var horAction = Mathf.FloorToInt(actions.DiscreteActions[0]);
20           var verAction = Mathf.FloorToInt(actions.DiscreteActions[1]);
21           Vector2 mov = Vector2.zero;
22           if (horAction == 1)
23           {
24              // right
25              dir.x = -1;
26              mov.x = -1;
27              animator.SetInteger("Direction", 3);
28           }
29           else if (horAction == 2)
30           {
31              // left
32              dir.x = 1;
33              mov.x = 1;
34              animator.SetInteger("Direction", 2);
35           }
36           if (verAction == 1)
37           {
38              // front
39              dir.y = 1;
40              mov.y = 1;
41              animator.SetInteger("Direction", 1);
42           }
43           else if (verAction == 2)
44           {
45              // back
46              dir.y = -1;
47              mov.y = -1;
48              animator.SetInteger("Direction", 0);
49           }
50           dir.Normalize();
51
52           if (fieldOfView != null)
53           {
54              fieldOfView.SetAimDirection(dir);
55           }
56           animator.SetBool("IsMoving", mov.magnitude > 0);
57           GetComponent<Rigidbody2D>().velocity = speed * mov;
58        }
```

Programa 4.2 Classe do agente do inimigo. Parte 2

```
1   public override void Heuristic(in ActionBuffers actionsOut)
2   {
3      Vector2 mov = Vector2.zero;
4      var action = actionsOut.DiscreteActions;
5      if (Input.GetKey(KeyCode.A))
6      {
7         action[0] = 1;
8      }
9      else if (Input.GetKey(KeyCode.D))
10     {
11        action[0] = 2; ;
12     }
13
14     if (Input.GetKey(KeyCode.W))
15     {
16        action[1] = 1;
17     }
18     else if (Input.GetKey(KeyCode.S))
19     {
20        action[1] = 2;
21     }
22
23  }
24
25  public override void OnEpisodeBegin()
26  {
27     gameObject.transform.localPosition = startingPos;
28     GetComponent<SpriteRenderer>().sortingLayerName = startingLayer;
29  }
30
31  private void Reset()
32  {
33     gameObject.transform.localPosition = startingPos;
34     GetComponent<SpriteRenderer>().sortingLayerName = startingLayer;
35     EndEpisode();
36  }
37
38  private void OnCollisionEnter2D(Collision2D collision)
39  {
40     if (collision.gameObject.tag == "Player")
41     {
42        AddReward(1.0f);
43        Debug.Log("KILL");
44        enemyRef.Reset();
45        Reset();
46     }
47  }
```

4.1.2 Observações

Para tratar das observações de cada inimigo, foi utilizado um componente padrão da biblioteca ML-Agents, o *Ray Perception Sensor 2D*, além de já se adicionar ao agente como uma observação, ele permite que o agente detecte o ambiente em volta de si utilizando `Unity RayCasts`, que são basicamente raios invisíveis que colidem com o ambiente, e podem ser vistos na aba *Scene* do Unity.

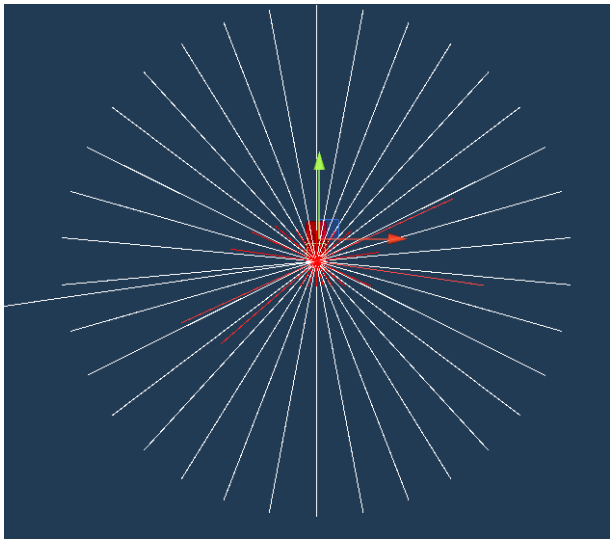

Figura 4.3: *Sensores saindo do agente.*

Foram configurados dois sensores diferentes, um para o ambiente e outro para detectar o protagonista exclusivamente:

4.1.3 Hiper-parametros

Os hiperparâmetros do modelo são parâmetros definidos antes do treinamento e que afetam o comportamento e o desempenho do modelo em questão. Algumas das configurações como o tamanho do lote (batch size), tamanho do buffer, taxa de aprendizado (learning rate), fator de desconto (gamma), número de épocas (num epoch), força de sinal de recompensa (reward signal strength), tamanho de codificação (encoding size), e assim por diante, foram dadas, porém há diversos hiper-parametros que podem ser implementados em um modelo.

O proposito desse tipo de ajuste é otimizar o desempenho do modelo, e diferentes valores podem levar a resultados totalmente discrepantes. Neste exemplo específico, há dois comportamentos definidos: o PlayerBehaviour e o EnemyBehaviour, ambos treinados com o algoritmo PPO mencionado anteriormente. Os hiperparâmetros específicos são definidos para cada comportamento, incluindo os parâmetros do modelo, do sinal de recompensa e do autojogo (self-play), entre outros.

De fato, os hiperparâmetros definidos nesse projeto são apenas um esboço inicial e podem não ser os ideais para o modelo em questão. A busca pelos hiperparâmetros corretos é um processo complexo e muitas vezes demanda tempo e esforço desproporcional ao propósito deste projeto, já que hã milhares de combinações de hiperparâmetros que podem

Programa 4.3 Hiper parametros do modelo do inimigo

```
behaviors:
  EnemyBehaviour:
    trainer_type: ppo
    hyperparameters:
      batch_size: 2048
      buffer_size: 20480
      learning_rate: 0.0003
      beta: 0.005
      epsilon: 0.2
      lambd: 0.95
      num_epoch: 3
      learning_rate_schedule: constant
    network_settings:
      normalize: false
      hidden_units: 512
      num_layers: 2
      vis_encode_type: simple
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
      curiosity:
            strength: 0.5
            gamma: 0.99
            encoding_size: 128
    keep_checkpoints: 5
    max_steps: 30000000
    time_horizon: 1000
    summary_freq: 10000
    self_play:
      save_steps: 50000
      team_change: 200000
      swap_steps: 4000
      window: 10
      play_against_latest_model_ratio: 0.5
      initial_elo: 1200.0
```

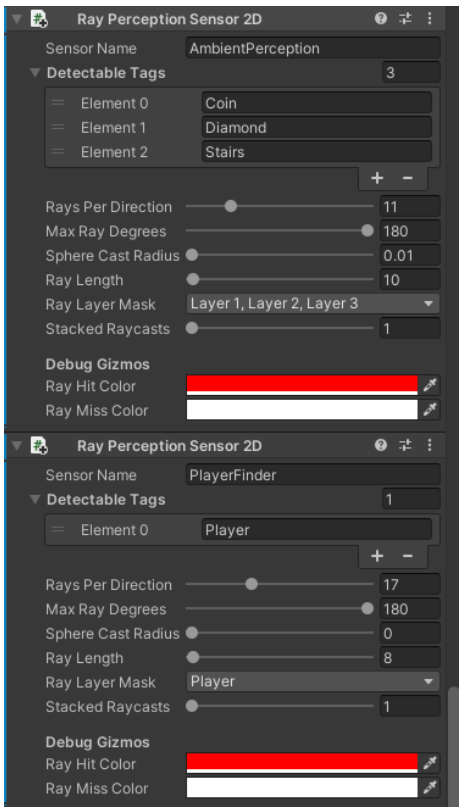

Figura 4.4: *Configuração de cada sensor.*

afetar significativamente o desempenho do modelo e encontrar os valores ideais em muitas vezes exige tentativa e erro.

É comum realizar vários experimentos para testar diferentes configurações de hiperparâmetros e avaliar o desempenho do modelo. Isso pode envolver a realização de testes em diferentes conjuntos de dados, ajustando as combinações de hiperparâmetros e avaliando os resultados. Existem técnicas de busca de hiperparâmetros que podem ajudar a automatizar esse processo, como a busca em grade (grid search) e a busca aleatória (random search), que permitem testar diferentes combinações de hiperparâmetros de forma mais eficiente. Porém, como mencionado anteriormente, esse tipo de ajuste foge da proposta deste projeto, e fica como melhoria para o futuro.

Capítulo 5

Conclusão

5.1 Resultados

Treinar um modelo de *Machine-Learning* requer centenas de milhares de interações, uma quantidade quase impossível de se obter através de treinamento com um humano. Para contornar esse obstáculo, foi criado um modelo similar ao do inimigo, porem simulando um jogador humano que busca coletar os itens do jogo. E os resultados estão representados abaixo

Figura 5.1: *No eixo Y: a recompensa cumulativa de cada agente. No eixo X o número de passos. (Inimigo em vermelho, protagonista em verde*

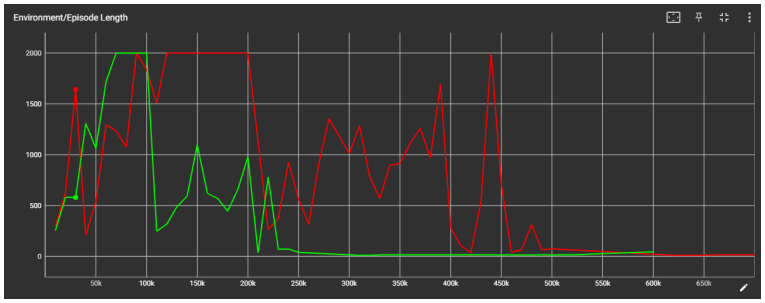

Figura 5.2: *No eixo Y: o tempo simulado para terminar cada episódio. No eixo X o número de passos. (Inimigo em vermelho, protagonista em verde*

É notável que após muitas sessões de treino, o modelo do inimigo conseguiu atingir o comportamento esperado, conseguindo pegar o jogador quase toda vez, porém é importante

notar que para haver impacto na maneira que o modelo se comporta foi necessário uma série de episódios de treino. Essa quantidade nem sempre é atingível com jogadores humanos, e o jogo precisaria ser extremamente popular para apresentar resultados significativos, o que é algo improvável de acontecer.

Portanto, podemos dizer que jogadores humanos não são uma fonte de dados eficiênte na coleta de dados para *Machine-Learning*, exceto em situações de larga escala, o que não é o caso do projeto.

Referências

BISOI, A K; DASH, S K; PATRA, S. Stock market prediction using unsupervised learning. In: IEEE. 2017 International Conference on Computational Intelligence and Communication Networks (CICN). [S.l.: s.n.], 2017. P. 1–6. Citado na p. 3.

CHAVES, F. **Engines de Jogos: Uma Introdução**. [S.l.]: Novatec, 2018. Citado na p. 5.

CONTRIBUTORS, PyTorch. **PyTorch Documentation**. [S.l.: s.n.], 2021. https://pytorch.org/docs/stable/index.html. Accessed on 2023-02-07. Citado na p. 6.

GOMES, Rosângela Aparecida et al. Working conditions and health of security guards. **Revista Brasileira de Saúde Ocupacional**, v. 36, n. 120, p. 53–61, 2011. Citado na p. 1.

JHSF. **Shopping Cidade Jardim**. [S.l.: s.n.], 2023. Disponível em: <https://jhsf.com.br/shopping-cidade-jardim/>. Citado na p. 10.

KNIGHTSCOPE. **K5 Robot**. [S.l.: s.n.], 2023. https://www.knightscope.com/k5/. Citado na p. 1.

MNIH, Volodymyr et al. Asynchronous methods for deep reinforcement learning. **International Conference on Machine Learning**, p. 1928–1937, 2016. Citado na p. 7.

OLIVEIRA, L. **Desenvolvimento de jogos eletrônicos com engine**. [S.l.]: Elsevier, 2010. Citado na p. 5.

OPENAI. **How to Train Your ML-Agent**. [S.l.: s.n.], 2021. https://www.youtube.com/watch?v=_9aPZH6pyA8. Accessed on: 2022-02-10. Citado na p. 15.

PATEL, Avni; SINGH, Rupinder. The impact of job stress on the physical and mental health of security guards. **Occupational Health and Safety**, v. 88, n. 4, p. 345–352, 2019. Citado na p. 1.

RAJPURKAR, Pranav et al. CheXNet: Radiologist-Level Pneumonia Detection on Chest X-Rays with Deep Learning. **arXiv preprint arXiv:1711.05225**, 2017. Citado na p. 3.

SCHULMAN, John et al. Proximal Policy Optimization Algorithms. **arXiv preprint arXiv:1707.06347**, 2017. Citado na p. 7.

STONE, J. **Unity Game Development Essentials**. [S.l.]: Packt Publishing, 2017. Citado na p. 5.

TECHNOLOGIES, Unity. **ML-Agents**. [S.l.: s.n.], 2021. Disponível em: <https://learn.unity.com/product/ml-agents>. Citado na p. 3.

_____._____. [S.l.: s.n.], 2021. Disponível em: <https://github.com/Unity-Technologies/ml-agents>. Citado na p. 6.

_____. **Unity Machine Learning Agents Toolkit**. [S.l.: s.n.], 2021. Disponível em: <https://github.com/Unity-Technologies/ml-agents-toolkit>. Citado na p. 6.

TÜRKER, Selahattin et al. The Relationship between Working Conditions and Mental Health among Security Guards. **International Journal of Occupational Safety and Ergonomics**, v. 22, n. 1, p. 1–9, 2016. Citado na p. 1.