Adaptative Game Boss

1st Nícolas Pereira
Unifesp
São José dos Campos, Brazil
nicolas.novaes@unifesp.br

2nd Thomas Lincoln
Unifesp
Caçapava, Brazil
lincoln.silva@unifesp.br

I. Introdução

O desenvolvimento de um modo de dificuldade adaptativo para jogos apresenta um desafio interessante e prático, com potencial para melhorar significativamente a experiência do jogador. Ao invés de oferecer apenas opções fixas de dificuldade fácil ou difícil, essa abordagem tem em vista ajustar a jogabilidade conforme as habilidades individuais do usuário em tempo real. Essa flexibilidade permite que jogadores de todos os níveis de habilidade desfrutem de uma experiência equilibrada e personalizada, mantendo a integridade da visão original do desenvolvedor em relação aos desafios do jogo.

O cerne da proposta reside na capacidade da IA em analisar o desempenho do jogador e adaptar-se conforme necessário, eliminando a necessidade de escolhas pré-definidas de dificuldade. Isso não só amplia o apelo do jogo para um público mais amplo, mas também enriquece a experiência ao garantir que a dificuldade evolua de forma orgânica, refletindo as habilidades do jogador.

O interesse deste projeto está na sua capacidade de equilibrar as expectativas dos jogadores com a visão do desenvolvedor, oferecendo uma experiência personalizada e dinâmica. Ao mesmo tempo, em que promove a acessibilidade e a inclusão, essa abordagem representa um avanço prático no design de jogos e na aplicação da IA na indústria de entretenimento digital.

II. Motivação

Compreender por que este trabalho é relevante é importante para entender seu propósito. Desenvolver um modo de dificuldade adaptativo para jogos surge da necessidade de melhorar a experiência do jogador. Os modos de dificuldade tradicionais muitas vezes não atendem às necessidades de todos os jogadores, deixando alguns entediados com desafios muito fáceis e outros frustrados com dificuldades muito altas. A motivação central é encontrar uma solução que se adapte dinamicamente ao nível de habilidade de cada jogador.

Ao criar esse modo de dificuldade adaptativo, o objetivo é garantir que todos os jogadores possam desfrutar do jogo, independentemente de sua habilidade. Isso tornaria a experiência mais inclusiva e satisfatória para todos. Além disso, usar inteligência artificial para ajustar a dificuldade em tempo real pode melhorar ainda mais essa adaptação, mantendo a visão original do desenvolvedor.

Identify applicable funding agency here. If none, delete this.

Em resumo, a motivação por trás deste trabalho é simplesmente fazer com que mais pessoas possam se divertir jogando, sem ficarem frustradas com desafios muito fáceis ou muito difíceis. É uma maneira prática de tornar os jogos mais acessíveis e divertidos para todos.

III. Conceitos Fundamentais

Para resolver o problema de ajustar a dificuldade do jogo sem exigir que o jogador selecione diretamente seu nível de dificuldade, é possível adotar abordagens como as propostas no modelo desenvolvido pelo Adaptive Game Systems [1]. Nesse modelo, o jogo monitora continuamente as ações realizadas pelo jogador para criar um perfil ou persona do jogador. Com base nesse perfil, o jogo pode então ajustar sua dificuldade conforme as características desse perfil específico. Os perfis são selecionados com base nas decisões e na performance do jogador.

Soluções como essa visam observar o jogador por meio de várias perspectivas, como o tempo necessário para concluir um objetivo, suas preferências de trajeto e sua eficácia ao derrotar inimigos. Para isso, são empregadas diversas técnicas de classificação e rotulagem. Esses métodos culminam na criação de um modelo probabilístico para prever os resultados prováveis de eventos no jogo. Essa abordagem visa evitar que o jogador entre em loops que comprometam o ajuste da dificuldade, garantindo uma experiência de jogo mais fluida e envolvente.

Alguns exemplos de jogos que empregam esses modelos incluem Mario Kart [4], Half-Life [5] e Resident Evil 4 [6]. Nos dois últimos, a adaptabilidade é evidenciada através dos itens concedidos ao jogador: caso o desempenho seja baixo e a saúde esteja reduzida, o jogo aumentará as chances de encontrar itens de recuperação de vida. No Mario Kart, quanto mais distante da primeira posição o jogador estiver, melhores serão os itens recebidos para auxiliá-lo a se recuperar e a competir.

Outro grupo de abordagens se baseia em aspectos emocionais, como descrito no trabalho referenciado em [2]. Nesse tipo de abordagem, os parâmetros para o ajuste da dificuldade são os estados emocionais do jogador, monitorados por meio de sinais fisiológicos que podem indicar seu nível de ansiedade. Com essas informações, é possível adaptar o nível de dificuldade conforme as intenções do game designer. Essa abordagem está alinhada com o conceito de Flow, no qual se tem em vista evitar que o jogador sinta que o jogo está muito

fácil ou muito difícil, proporcionando uma experiência de jogo mais envolvente e satisfatória.

Por fim, há os métodos baseados em aprendizado, exemplificados no trabalho referenciado em [3]. Essas abordagens buscam criar um método distinto para ajustar a dificuldade, adaptando os comportamentos dos agentes de IA por meio de algoritmos de aprendizado, como o aprendizado por reforço. Dessa forma, o game designer fica menos responsável por ajustar meticulosamente a dificuldade do jogo. Nesse tipo de abordagem, são geralmente empregadas técnicas de aprendizado offline, nas quais o modelo é treinado intensivamente, ou então são coletados dados de jogo de jogadores humanos em quantidade suficiente para o treinamento.

Um exemplo popular dessa abordagem é encontrado no jogo Resident Evil 4 [6], onde o sistema se adapta com base na precisão dos tiros do jogador. Se o jogador acertar repetidamente a cabeça dos inimigos com precisão, estes começarão a desviar dos tiros ou a se proteger com as mãos, uma vez que um tiro na cabeça é a maneira mais eficaz de derrotá-los nesse jogo.

Também é necessário discutir sobre os dois algoritmos mais usados quando se trata de treinamento de modelos para jogos. Visando o uso da biblioteca ML-Agents [7], o algoritmo padrão é o PPO (Proximal Policy Optimization) [8] e outra alternativa é o SAC (Soft Actor-Critic) [9], juntos esses dois provém duas possíveis alternativas no treino de modelos.

Um dos algoritmos mais usados e bem-sucedidos em Aprendizado por Reforço (RL) é o PPO. Ele é uma evolução de métodos anteriores como TRPO (Trust Region Policy Optimization). Encontrar um equilíbrio entre simplicidade e desempenho, mantendo a estabilidade durante o treinamento, o que é um desafio comum em algoritmos de RL, é a principal ideia por trás do PPO.

O objetivo do PPO é treinar uma política (policy), o qual é uma função que mapeia os estados de um ambiente para ações. A política é representada por uma rede neural, e o algoritmo visa otimizar essa política para maximizar o retorno esperado (soma das recompensas) ao longo do tempo.

Esse algoritmo trabalha com duas versões da mesma política ao mesmo tempo, uma é a versão atual dela que sofre otimizações e a outra é uma versão congelada que serve como comparativo. Outro ponto importante é que esse algoritmo implementa uma função de perda mais balanceada, chamada de Clipped Surrogate Objective [10] ela faz com que as mudanças feitas de uma geração para outra fiquem em um certo limite, assim, ajudando para a estabilidade do modelo.

O modelo aprende da seguinte forma, ele primeiro interage com o ambiente coletando informações do mesmo, coletando dados sobre ações, recompensas e estados futuros. Após isso, ele calcula o quão vantajosa é uma ação tomando como comparação o resultado esperado. A política é aplicada visando a minimização da perda.

Concluindo, esse algoritmo foi criado buscando simplicidade e estabilidade, é um algoritmo generalista, que pode ser aplicado em vários casos diferentes.

Em contrapartida, também temos o SAC que é um que se destaca por ser eficaz em problemas com espaços de ação contínuos. O SAC é uma melhoria de outros métodos fora da política, como o DDPG (Deep Deterministic Policy Gradient) [11]. Combina os conceitos de máxima entropia com aprendizado por reforço, tornando-o especialmente forte e estável em tarefas complexas.

O objetivo do SAC é desenvolver uma política que maximize tanto o retorno esperado (a soma das recompensas ao longo do tempo) quanto a entropia da política. Neste caso, a entropia estimula políticas mais "exploratórias", avaliando o grau de aleatoriedade nas ações do agente.

Esse algoritmo usa três redes neurais principais, a política essa rede neural, também conhecida como "actor", conecta estados para uma distribuição de ações. A política no SAC introduz aleatoriedade controlada em vez de selecionar uma ação específica.

E duas redes críticas estimam o valor-Q (Q-value), ou a recompensa esperada de um estado específico ao tomar uma ação específica. A estabilidade do treinamento aumenta e a variância diminui quando há duas avaliações em vez de uma.

O SAC otimiza a entropia política e a recompensa esperada, o que o torna um grande diferencial. O agente é incentivado a manter suas ações "exploratórias" ao aprender a maximizar as recompensas, conforme a ideia de máxima entropia. Isso ajuda a evitar que o agente se torne demasiado exploitativo. Isso pode significar que ele se prende a uma estratégia que pensa ser a melhor, mas que pode não ser a melhor em todo o mundo.

Três componentes principais compõem a função de perda do SAC:

Perda da Crítica (Q-loss): As críticas são atualizadas para reduzir a diferença entre o valor-Q previsto e o alvo, que foi calculado com base em recompensas e valores futuros.

Perda da política, ou perda da política: a política é modificada para maximizar o valor-Q e a entropia. Isso é feito modificando os parâmetros da política para as ações serem tanto exploratórias quanto recompensadoras.

Atualização da Temperatura: A temperatura regula a compensação entre exploração (alta entropia) e exploração (alta recompensa). Durante o treinamento, o SAC ajusta automaticamente essa temperatura para atingir um equilíbrio ideal.

O SAC consegue reutilizar experiências anteriores armazenadas em um buffer de repetição para atualizações, pois um algoritmo fora da política. Em termos de eficiência, permite que o algoritmo aprenda mais rapidamente com menos interações diretas com o ambiente.

As tarefas que exigem espaços de ação contínuos, como controle robótico, manipulação de objetos e jogos nos quais as ações podem ser ajustadas com precisão, exigem que o SAC funcione. Sua maior vantagem inclui:

Explosão forte: A máxima entropia garante que o agente não caia em estratégias sub ótimas.

Eficiência amostral: Como é fora da política, o SAC pode aprender com a reutilização de experiências anteriores.

Estabilidade do treinamento: A utilização de duas críticas e a atualização automática da temperatura garantem um treinamento estável, mesmo em tarefas desafiadoras.

Esses dois algoritmos formam as opções mais usadas na biblioteca ML-agents a qual será usada para o desenvolvimento desse projeto de pesquisa.

Também bale ressaltar a estratégia min/máx ou minimax [**?**], é um método usado principalmente em teoria dos jogos e inteligência artificial, especialmente em problemas de decisão e jogos de soma zero, como xadrez, onde dois jogadores estão em oposição direta. O objetivo é minimizar a perda máxima possível (daí o nome "minimax").

O Minimax tem um princípio relativamente simples, dos dois jogadores, um, tenta maximizar seus ganhos enquanto o outro, tenta reduzir os ganhos oponentes. O jogador que maximiza em um jogo é chamado de "Max" e o jogador que minimiza é chamado de "Min".

Uma árvore pode ser usada para representar todas as jogadas possíveis em um jogo, com cada nó representando um estado do jogo. As folhas da árvore representam o resultado do jogo.

Cada estado final (folha) da árvore recebe um valor numérico que indica quão bom ou ruim é para o jogador Max. Por exemplo, +1 pode significar uma vitória para Max, -1 uma derrota e 0 um empate.

As folhas transmitem o valor para o nó raiz da árvore.

Ao jogar, Max escolherá o filho com o valor máximo para maximizar seu ganho. Ao jogar, Min escolherá o movimento que reduz o ganho de Max, escolhendo o filho com o valor mínimo.

Por exemplo, imagine um jogo muito simples onde Max e Min podem escolher entre dois movimentos. A árvore pode ser representada seguindo o exemplo abaixo.

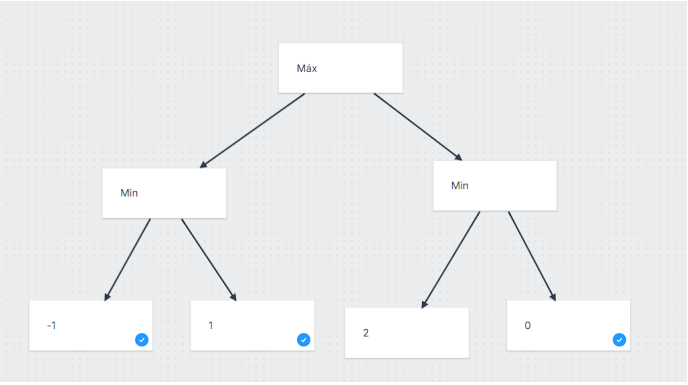

Fig. 1. modelo da árvore

- Na primeira jogada, Max tem duas escolhas. Ele pode ir para a esquerda ou para a direita.
- Se Max for para a esquerda, Min escolherá entre 1 e -1. Como Min quer minimizar o ganho de Max, ele escolherá -1.
- Se Max for para a direita, Min escolherá entre 2 e 0, então ele escolherá 0.
- Max, então, compara as opções que Min deixou para ele e escolherá a que maximiza seu ganho. Ele escolherá a esquerda porque -1 é maior que 0.
- Portanto, Max seguirá para a esquerda, mesmo que a escolha inicial de Min minimizasse seu ganho.

O algoritmo Minimax pode ser aplicado em uma variedade de contextos além dos jogos:

AI para jogos de tabuleiro, como xadrez e damas; planejamento e decisão, quando dois agentes ou forças se opõem em um problema; negociações, em que cada parte se esforça para obter o maior benefício possível e minimizar o do oponente.

Em situações mais complexas, a variação do Minimax Alpha-Beta Pruning é usada para otimizar a eficiência do algoritmo, eliminando a necessidade de explorar algumas subárvores da árvore de decisões, reduzindo drasticamente o tempo de cálculo.

IV. Objetivo

O objetivo deste projeto é desenvolver uma solução para adaptar a dificuldade dos jogos, buscando alcançar um equilíbrio individualizado para cada jogador. Isso permitirá ao game designer escolher como deseja que todos experimentem o jogo. Por exemplo, se o designer preferir que o jogo seja mais desafiador, ele poderá ajustar o modelo para que sempre busque aumentar a dificuldade para todos os jogadores. Também temos como objetivo testar o treino em diferentes algoritmos e ver como o modelo irá se comportar no mínimo/máximo.

V. Metodologia Experimental

A. O jogo

O jogo será desenvolvido na plataforma Unity como um jogo de plataforma 2D, apresentando uma jogabilidade inicial focada em um confronto entre um jogador e um chefe. O chefe será representado por um olho que precisa encostar no jogador repetidas vezes até reduzir sua vida a zero. Por outro lado, o jogador deverá atirar no chefe até esgotar a vida dele. Ambos os personagens seguirão estratégias baseadas no algoritmo Minimax, proporcionando uma dinâmica de combate onde ambos estarão aprendendo.

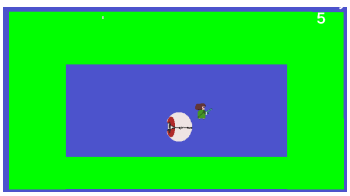

Fig. 2. print do jogo

No jogo, um jogador e um chefe se enfrentam na mesma arena. Quando o chefe vence, as paredes ficam verdes, e quando ele perde, as paredes ficam vermelhas, facilitando a visualização durante o treinamento. Essa mudança de cor foi implementada para simplificar o desenvolvimento. Tanto o jogador quanto o chefe se movem de acordo com um modelo,

resultando em movimentos tecnicamente aleatórios. À medida que aprendem, eles desenvolvem estratégias para derrotar um ao outro.

B. O código

O código todo é feito usando quatro scripts diferentes, sendo eles dois para o chefe: "Boss" [13] e "BossAgent" [14], e dois para o player: "Player" [15] e "PlayerController" [16]. A seguir uma explicação básica de cada um deles.

1) Boss: O script "Boss" controla um inimigo em um jogo de plataforma 2D, definindo suas ações, a maneira como ele interage com o jogador e como ele aprende com essas interações. O código é estruturado para o Chefe reagir ao ser atingido pelo jogador. Além disso, ele inclui uma interface de usuário simples que exibe a quantidade de vida do Chefe.

O script começa com a declaração de várias variáveis públicas e privadas, que configuram as propriedades do Chefe como a vida (HP) do Boss.

- Vida e Dano: bossHP determina quantos pontos de vida o Chefe tem. O atributo canTakeDamage é usado para controlar quando o Chefe pode receber dano.
- Sistema de Penalização: escalaDePenalizacao, countErros, e countAcertos ajudam a controlar a dificuldade, ajustando recompensas e penalidades baseadas nas ações do Chefe.

Uma variedade de métodos interdependentes estabelecem o comportamento do chefe e sua interação com o jogador e o ambiente. A lógica do Chefe começa invocando o método Start(), quando o jogo começa. Esse método inicializa o agente do chefe, os quais são uma instância de BossAgent. O texto da interface do usuário que mostra a vida do Chefe é atualizado pela primeira vez usando esse mesmo método.

O método OnTriggerEnter2D (Collider2D other) é usado quando a espada do jogador atinge o Chefe. Essa técnica verifica se o Chefe foi atingido por um objeto com a tag "Sword". Se ele pode receber dano imediatamente, como controlado pela variável "canTakeDamage", a vida do chefe é reduzida. Atualmente, a interface é modificada para incorporar a nova quantidade de vida. O método Die() é chamado se a vida do chefe chega a zero, indicando a derrota do chefe. O método EndEpisode() registra o evento e dá uma recompensa ao agente antes de reiniciar o jogo.

O método EndEpisode() fecha o episódio de treinamento do agente e reinicia a cena, permitindo que o ciclo de aprendizado continue. O método DamageCooldown() é usado durante o jogo para evitar que o Chefe seja atingido repetidamente em um curto período. Esse método permite que o chefe tenha um período de invulnerabilidade após ser atingido, reduzindo a frequência de danos.

E por fim o método UpdateBossHealthText() atualiza a interface do usuário com a vida atual do Chefe, garantindo que o jogador esteja sempre ciente da quantidade de dano que infligiu ao Chefe.

O script integra o Unity ML-Agents, permitindo que o Chefe aprenda com as batalhas. Recompensas são dadas ao Chefe quando ele acerta o jogador (bossAgent.AddReward(1.0f)) e penalidades são aplicadas quando ele erra (bossAgent.AddReward(-1.0f * escalaDePenalizacao)).

2) BossAgent: Usando o Toolkit ML-Agents e o Unity, esse código implementa um agente de aprendizado de máquina chamado BossAgent. Ele pode controlar o comportamento de um Chefe em um ambiente de jogo. Este agente aprende com o tempo por meio de recompensas e penalidades com o jogador e o ambiente.

O método Start() é chamado quando o jogo é iniciado. Esse método inicializa o Rigidbody2D do agente, permitindo que o Chefe se mova fisicamente na cena. Além disso, as primeiras atualizações da interface do usuário que mostram a saúde do Chefe e do jogador foram realizadas. A variável bossHP, que foi inicialmente configurada para 10, representa a saúde do chefe.

O método OnEpisodeBegin() é chamado quando um novo episódio começa. Este método reposiciona o chefe e o jogador na primeira posição da cena, zerando a velocidade do chefe. A saúde do jogador e do chefe é reiniciado para seus valores originais, e a interface do usuário é atualizada para refletir esses valores. Além disso, a posição do jogador é alterada aleatoriamente em um determinado espaço e o timer é reiniciado para zero.

O agente usa o método CollectObservations (sensor de vetor) para coletar observações do ambiente durante o episódio. Este método coleta as posições do chefe e do alvo, que podem ser as posições do jogador ou de outro objeto crucial, e então essas informações são enviadas para o modelo de aprendizado de máquina. O modelo usa essas informações para tomar decisões sobre o que o chefe deve fazer.

O método OnActionReceived(ActionBuffers actions) escolhe as ações do Chefe. Este método usa os valores contínuos dos movimentos nos eixos X e Y que foram determinados pelo modelo. Para movê-lo na cena, esses valores são aplicados ao Rigidbody2D do Chefe. Cada quadro durante o episódio recebe um temporizador. Se o tempo exceder um limite, conhecido como TimeLimit, o agente será penalizado por tentar tomar decisões mais rapidamente.

O método Heuristic (em ActionBuffers actionsOut) permite que o desenvolvedor controle manualmente o Chefe usando as teclas de entrada ("Horizontal" e "Vertical") se o jogo estiver sendo executado em modo de depuração ou sem um modelo treinado. É útil para ajustes e testes.

Quando o Chefe colide com outros objetos na cena, o método OnTriggerEnter2D (Collider2D other) é usado. As seguintes ações são tomadas quando ele colide com um objeto:

- Se o chefe bater em um objeto de objetivo, ele recebe uma recompensa e o chão muda de cor para verde, indicando sucesso. Para determinar se o episódio deve ser encerrado, o método CheckEndCondition() é usado.
- Se o chefe bater em uma parede, ele perde um ponto de vida (bossHP), a cor do chão muda para vermelho e recebe uma penalidade reduzida. Para determinar se o episódio deve ser encerrado, uma vez mais, CheckEndCondition() é chamado.

- Se o Chefe for atingido por uma espada (Sword), ele causa dano ao jogador, atualiza a interface do usuário e recebe uma recompensa. O episódio é encerrado se a vida do jogador ou do chefe acaba.
- Quando o chefe colide com um feixe (Beam), ele perde um ponto de vida e também é punido. O episódio termina se a vida do Chefe acabar.

O método ChangeFloorObjectsColor (Color color) pode alterar a cor de certos objetos no chão, representados por floorObjects, de acordo com um evento que ocorreu. Esses objetos podem mudar de cor conforme o evento, como o sucesso na consecução de um objetivo ou uma penalidade por colidir com uma parede. Isso dá feedback visual sobre o jogador e o chefe.

A função UpdateHealthUI() é responsável por atualizar a interface do usuário e exibir a saúde atual do chefe e do jogador. Sempre que a saúde de um deles muda, essa função é chamada.

Por último, mas não menos importante, o método CheckEndCondition() determina se o episódio deve ser encerrado. Isso acontece quando a vida do Chefe ou do jogador chega a zero. Após isso, o método EndEpisode() é chamado, o que encerrará o episódio e dá ao agente a capacidade de iniciar um novo ciclo de aprendizado com base nas experiências que ele acumulou.

3) Player: Esse código executa as ações do jogador controlado pelo computador. Este script especifica as ações do jogador, incluindo movimento, mudança de direção, tiro de projéteis e reação a danos.

O atributo playerHP, o qual é a quantidade de pontos de vida do jogador, é determinado pelo jogador e começa com 5. Quando o jogador sofrer dano, esse valor diminui. O Rigidbody2D do jogador é inicializado no método Start(), o que permite que ele seja movido fisicamente na cena. Isso também é conhecido como a função ChangeDirection(), que determina de forma aleatória a direção inicial do movimento do jogador.

Duas ações principais ocorrem no método Update(), chamado a cada quadro: mudança de direção e disparo de projéteis. Conforme o tempo passa, o temporizador de mudança de direção aumenta. Quando atingir um valor de mudança de direção de intervalo de 1,5 segundos, o jogador muda sua direção de movimento para uma direção aleatória, o temporizador é reiniciado. Semelhantemente, o tempo de disparo é aumentado. Quando o tempo de disparo chega a um segundo, o jogador dispara um projétil em uma das quatro direções cardinais (cima, baixo, esquerda ou direita).

O método MovePlayer() ajusta a direção de movimento do jogador ao Rigidbody2D, ajustando a velocidade apropriadamente. A velocidade do jogador pode ser calculada multiplicando o vetor de movimento por dois.

O método ChangeDirection() muda a direção do movimento do jogador. A direção é determinada normalizando dois números aleatórios que oscilam entre -1 e 1. Isso obriga o jogador a se mover em diferentes direções a cada intervalo de tempo.

O método ShootProjectile() é usado quando o jogador dispara um projétil. Este método permite que um novo objeto de projétil seja inserido na posição e rotação do jogador. O projétil é então lançado em uma das quatro direções possíveis (cima, baixo, esquerda ou direita). A velocidade é determinada pelo vetor de direção multiplicado por um fator de cinco. Esse projétil consegue se mover na cena e interagir com outros objetos.

O método IsOutsideSpawnArea() verifica se o jogador excedeu os limites da área de spawn das variáveis spawnAreaMin e spawnAreaMax. O método TeleportBackToArea() reposiciona o jogador nos limites da área se ele sair dela, selecionando uma nova posição nas coordenadas permitidas.

O método TakeDamage permite que o jogador sofra dano. Esse método reduz o valor do jogadorHP em 1. Além disso, o método UpdateHealthUI() do objeto BossAgent é usado para atualizar a interface do usuário. Caso os pontos de vida do jogador cheguem a zero, o método Die() é usado. Isso envia uma mensagem de derrota para o console e verifica com o método CheckEndCondition() do BossAgent se o episódio atual do jogo deve terminar.

4) Player Controller: O comportamento de um jogador em um jogo 2D é controlado pelo script PlayerController. O jogador pode se mover horizontalmente, pule e atacar com uma arma. No eixo X, os valores mínimos e máximos limitam o movimento. O script inclui tanto movimento aleatório, criado por uma co-rotina que faz o jogador mover-se, pular e atacar aleatoriamente em intervalos regulares ou permite movimento controlado quando o movimento é controlado pelo usuário, gerado por entradas do teclado.

Se o jogador estiver no chão, ele pode saltar se pressionar um botão ou se a co-rotina decidir que deve atacar. O script move a arma para simular um golpe enquanto o jogador ataca.

Quando o jogador colide com um objeto marcado como "Ground", ele é considerado "no chão". Objetos com as marcas "Beam" ou "Boss" também podem atingir o jogador, diminuindo sua vida (playerHP). Se a vida do jogador estiver zero, ele "falece", retornando à sua posição original e restaurando seu valor original.

C. Ml-Agents

O ML-Agents (Machine Learning Agents) é uma ferramenta da Unity que permite integrar aprendizado de máquina em jogos e simulações, permitindo que personagens ou agentes em um jogo aprendam comportamentos complexos por meio de treino com reforço.

O ML-Agents foi usado para treinar o BossAgent para interagir com o ambiente e o jogador (Player). O BossAgent observa a posição e a posição do jogador e toma decisões sobre movimento e comportamento.

O método OnActionReceived processa essas escolhas e permite que o agente escolha como se mover no ambiente. O agente recebe recompensas ou penalidades durante o treinamento dependendo de suas ações, como atingir um jogador ou colidir com paredes. Com o passar do tempo, ele aprende

a usar a melhor maneira de agir para atingir metas específicas, como derrotar o jogador ou sobreviver por mais tempo.

D. Exploration-exploitation dilemma

Um conceito fundamental no aprendizado por reforço é o dilema de exploration e exploitation [19]. Ele fala sobre a tensão entre os dois métodos: explorar novas oportunidades para encontrar melhores opções (exploração) e usar as opções que já conhecemos que oferecem recompensas garantidas (exploração). O desafio é encontrar um equilíbrio entre esses dois comportamentos, pois explorar demais pode resultar em desperdício de recursos em opções ineficazes, enquanto explorar de menos pode resultar em perdas de oportunidades valiosas de melhora.

Um problema com o nosso modelo de otimização de políticas próximas (PPO) foi a excesso de exploitation. Após muitas tentativas, o modelo "aprendeu" que permanecer em um canto do ambiente tinha recompensas melhores. Esse comportamento é um exemplo típico de exploitation no qual o agente explora ao máximo uma estratégia que oferece recompensas constantes. No entanto, ao fazer isso, ele parou de tentar métodos alternativos que poderiam melhorar ainda mais o desempenho.

O comportamento do modelo de permanecer neutro mostra um desequilíbrio no dilema exploração-explotação. O modelo decidiu usar uma solução sub ótima como a melhor opção disponível. Isso é comum em sistemas de aprendizado por reforço, onde o agente aprende uma estratégia com recompensa garantida e deixa de procurar melhorias; isso pode ser prejudicial em ambientes mais complexos ou dinâmicos.

E. Aplicação do Minimax no projeto

As recompensas e penalidades foram cuidadosamente ajustadas para guiar o aprendizado dos agentes durante o treinamento. Ao acertar o jogador com projéteis ou combater diretamente com ele, o líder recebia recompensas favoráveis. Por outro lado, quando o Chefe colidia com paredes ou não conseguia derrotar o Player no tempo limitado, eram aplicadas penalidades, forçando-o a aprimorar seu controle e ataques. Além disso, a sobrevivência sem sofrer danos foi fortemente recompensada para o jogador, o que o encorajou a agir de maneira astuta e estratégica. Ao ser atingido ou colidir com obstáculos, punições foram aplicadas, reforçando a importância de evitar perigos e utilizar o ambiente a seu favor. A estrutura de recompensas e penalidades garantiu que cada agente desenvolvesse suas habilidades de acordo com seus objetivos: o Chefe se concentrando em maximizar o dano e o Player em minimizar os riscos e prolongar sua sobrevivência.

Para orientar o aprendizado dos agentes, as recompensas e penalidades foram ajustadas. O Chefe recebeu recompensas por acertar o jogador e encorajar golpes agressivos, além de ser penalizado por colidir com paredes ou não derrotar o jogador a tempo, o que o tornou mais produtivo. O jogador recebeu recompensas por sobreviver sem sofrer dano e usar movimentos evasivos. Se for atingido ou confrontado com obstáculos, também foi penalizado, aumentando a necessidade de evitar perigos. Isso permitiu que o jogador aprimorara suas habilidades de sobrevivência e o Chefe aprimorara seus ataques.

VI. O QUE SERÁ ENTREGUE NO FINAL?

No final deste projeto, será fornecida uma documentação, incluindo os algoritmos utilizados e os métodos de implementação. Além disso, serão mostrados gráficos mostrando o desempenho de cada um dos dois algoritmos implementados (PPO e SAC), os gráficos seguiram o modelo fornecido pelo TensorBoard [17], essa biblioteca nos permite visualizar as recompensas ganhas e a Loss de cada modelo ao longo do tempo.

A. Gráficos PPO

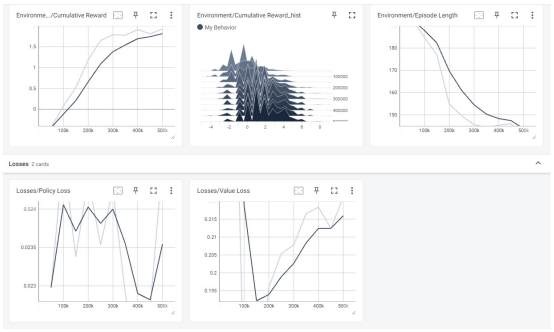

Fig. 3. gráficos do PPO

1) Environment/Cumulative Reward: Esse gráfico mostra as recompensas acumuladas ao longo do tempo. O número de etapas ou episódios de treinamento é representado por eixos horizontais. A linha clara, que pode ser o valor alvo ou a média, representa o desempenho desejado do agente, enquanto a linha escura representa o desempenho do agente atual.

2) Environment/Cumulative Reward_hist: A distribuição das recompensas acumuladas ao longo do treinamento é mostrada neste gráfico. A frequência das recompensas acumuladas em diferentes estágios do treinamento é representada pelas curvas aqui.

3) Environment/Episode Length: A duração dos episódios ao longo do tempo é mostrada neste gráfico. A diminuição do tempo de duração dos episódios pode indicar que o agente está aprendendo a maximizar sua recompensa ou atingir seu objetivo de forma mais eficaz, encerrando os episódios mais rapidamente.

4) Losses/Policy Loss: A perda relacionada à política do agente durante o treinamento é representada neste gráfico. Embora flutuações na perda sejam comuns, elas devem diminuir com o tempo, indicando que o agente está melhorando sua política.

5) Losses/Value Loss: A perda relacionada à função de valor durante o treinamento é representada neste gráfico. A função de valor determina quão útil é para o agente uma situação ou estado específico para suas decisões.

B. Gráficos Finais

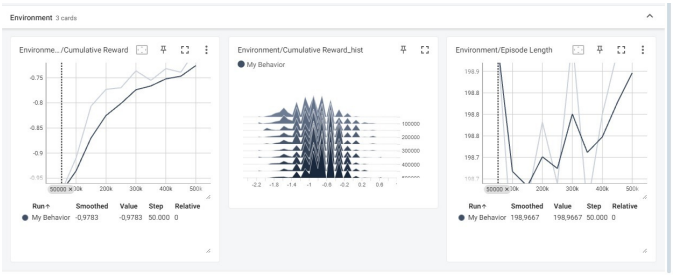

Fig. 4. Gráfico final PPO - 1

Fig. 5. Gráfico final PPO - 2

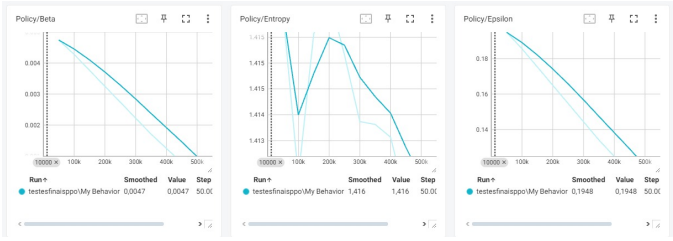

Fig. 6. Gráfico final PPO - 3

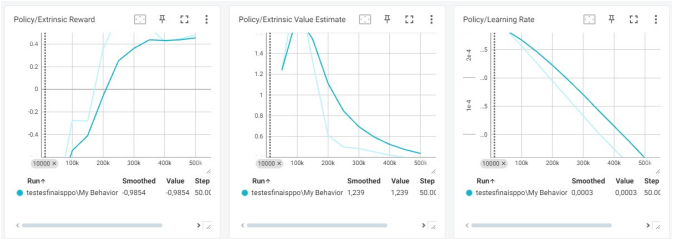

Fig. 7. Gráfico final PPO - 4

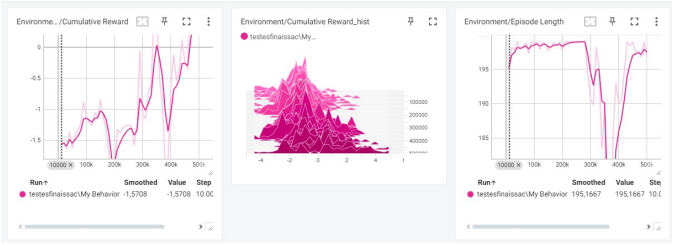

Fig. 8. Gráfico final SAC - 1

Fig. 9. Gráfico final SAC - 2

Fig. 10. Gráfico final SAC - 3

Fig. 11. Gráfico final SAC - 4

1) Conclusões a serem consideradas: O desempenho do agente mostra uma tendência ascendente constante no gráfico de prêmio cumulativo PPO, com o valor cumulativo do prêmio quase zero. Isso indica que, embora inicialmente tenha um valor de recompensa negativo, o PPO está gradualmente aprendendo a melhorar seu desempenho de forma mais estável. O gráfico de reembolso cumulativo já mostra maior variabilidade e flutuações mais marcantes no SAC. O valor cumulativo de recompensa no SAC começou a piorar, mas finalmente melhorou significativamente, demonstrando um comportamento mais instável durante o treinamento.

Além disso, as diferenças notáveis são mostradas no gráfico de duração do episódio. O PPO mostra um aumento gradual na duração dos episódios com o tempo, indicando que o agente está adquirindo habilidades para sobreviver por mais tempo. Por outro lado, o SAC oscila mais frequentemente durante os episódios, o que pode indicar que o agente está testando várias abordagens antes de estabilizar seu comportamento.

As curvas de perda (Q1 e Q2 Loss) no SAC mostram variações significativas no início do treinamento, mas eventualmente convergem para valores mais baixos. Isso indica que o modelo está aprendendo, embora de forma menos estável. Ao contrário, o PPO apresenta uma perda que diminui de forma mais rápida e uniforme, com menos oscilações, indicando maior estabilidade e convergência mais eficiente. Por fim, enquanto o SAC pode ser mais adequado para ambientes que exigem exploração mais dinâmica, o PPO se destaca por sua estabilidade e rapidez na convergência, tornando-se a opção mais confiável em situações em que essas qualidades são de grande importância.

REFERENCES

[1] CHARLES, D. et al. Player-centred game design: Player modelling and adaptive digital games. Proceedings of DiGRA 2005 Conference: Changing Views - Worlds in Play, 01 2005.

[2] LIU, C. et al. Dynamic difficulty adjustment in computer games through real-time anxiety-based affective feedback. Int. J. Hum. Comput. Interact., v. 25, n. 6, p. 506–529, 2009.

[3] TAN, C. H.; TAN, K. C.; TAY, A. Dynamic game difficulty scaling using adaptive behavior-based AI. IEEE Trans. Comput. Intell. AI Games, v. 3, n. 4, p. 289–301, 2011.

[4] https://www.nintendo.com/pt-br/store/products/mario-kart-8-deluxe-switch/

[5] https://store.steampowered.com/app/70/HalfLife/

[6] https://store.steampowered.com/app/254700/Resident_Evil_4_2005/

[7] https://docs.unity3d.com/Manual/com.unity.ml-agents.html

[8] https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html

[9] https://spinningup.openai.com/en/latest/algorithms/sac.html

[10] https://huggingface.co/learn/deep-rl-course/unit8/clipped-surrogate-objective

[11] https://spinningup.openai.com/en/latest/algorithms/ddpg.html

[12] https://www.sciencedirect.com/science/article/pii/S0888613X18302226

[13] https://github.com/NicolasComAcento/Projeto-IA/blob/main/Assets/Scripts/Boss.cs

[14] https://github.com/NicolasComAcento/Projeto-IA/blob/main/Assets/Scripts/BossAgent.cs

[15] https://github.com/NicolasComAcento/Projeto-IA/blob/main/Assets/Scripts/Player.cs

[16] https://github.com/NicolasComAcento/Projeto-IA/blob/main/Assets/Scripts/PlayerController.cs

[17] https://www.tensorflow.org/tensorboard?hl=pt-br

[18] https://en.wikipedia.org/wiki/Exploration-exploitation_dilemma

[19] https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0095693