UNIVERSIDADE FEDERAL FLUMINENSE

ÍCARO GOULART FARIA MOTTA FRANÇA

APRENDIZAGEM POR REFORÇO E POR IMITAÇÃO COM REDES NEURAIS APLICADA AO JOGO BOMBERMAN

NITERÓI

2019

UNIVERSIDADE FEDERAL FLUMINENSE

ÍCARO GOULART FARIA MOTTA FRANÇA

APRENDIZAGEM POR REFORÇO E POR IMITAÇÃO COM REDES NEURAIS APLICADA AO JOGO BOMBERMAN

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: COMPUTAÇÃO VISUAL.

Orientador:
Prof. Dra. ALINE MARINS PAES CARVALHO

Co-orientador:
Prof. Dr. ESTEBAN WALTER GONZALEZ CLUA

NITERÓI

2019

Ficha catalográfica automática - SDC/BEE
Gerada com informações fornecidas pelo autor

F814a França, Ícaro Goulart Faria Motta
Aprendizagem por Reforço e por Imitação com Redes Neurais Aplicada ao Jogo Bomberman / Ícaro Goulart Faria Motta França ; Aline Marins Paes Carvalho, orientador ; Esteban Walter Gonzalez Clua, coorientador. Niterói, 2019.
139 f. : il.

Dissertação (mestrado)-Universidade Federal Fluminense, Niterói, 2019.

DOI: http://dx.doi.org/10.22409/PGC.2019.m.12314180798

1. Aprendizagem por reforço. 2. Aprendizagem por imitação. 3. Redes neurais. 4. Representação de estado. 5. Produção intelectual. I. Marins Paes Carvalho, Aline, orientador. II. Walter Gonzalez Clua, Esteban, coorientador. III. Universidade Federal Fluminense. Instituto de Computação. IV. Título.

CDD -

Bibliotecária responsável: Fabiana Menezes Santos da Silva - CRB7/5274

ÍCARO GOULART FARIA MOTTA FRANÇA

APRENDIZAGEM POR REFORÇO E POR IMITAÇÃO COM REDES NEURAIS APLICADA AO JOGO BOMBERMAN

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: COMPUTAÇÃO VISUAL

Aprovada em Maio de 2019.

BANCA EXAMINADORA

Prof. Dra. Aline Marins Paes Carvalho (Orientadora), IC-UFF

Prof. Dr. Esteban Walter Gonzalez Clua (Orientador), IC-UFF

Prof. Dr. Anselmo Montenegro Antunes, IC-UFF

Prof. Dr. Luiz Chaimowicz, DCC-UFMG

Niterói

2019

Dedico esta dissertação à Força que sempre esteve comigo mesmo quando eu não estava com ela, à minha família, aos meus amigos e amigas, aos meus orientadores e a quem mais se interessar pelo assunto.

Agradecimentos

À minha família que está longe fisicamente mas que sempre me apoiou mesmo quando ninguém entendia quais eram os problemas de pesquisa que eu estava tentando solucionar.

À minha família adotiva de Niterói que me recebeu como filho e irmão. E que me ajudou a relaxar através de jogatinas: de cartas e jogos digitais.

À minha namorada que com seu jeito simples de ser, mesmo a distância, me ajudou a ficar mais tranquilo com a pressão do mestrado.

Aos meus amigos e amigas que me apoiaram presencialmente ou à distância.

Aos meus amigos e amigas da UFF que sempre me apoiaram e confraternizaram comigo em eventos, trilhas, bares, cinemas, paintball, rpg de mesa, kart e etc.

Aos meus parentes distantes que me receberam de portas abertas nos primeiros meses de mestrado.

À minha orientadora Aline Paes que sempre acreditou no meu potencial apesar de nem eu mesmo acreditar mais. Que como uma mestre Yoda junto com o co-orientador Esteban direcionavam a pesquisa para o rumo certo.

Ao meu orientador Esteban Clua que sempre me recebeu bem humorado e disposto.

Aos professores Luiz Chaimowicz e Rosilane Ribeiro da Mota por terem feito minhas cartas de recomendação para que eu pudesse concorrer ao processo seletivo de mestrado.

À uma amiga distante que no último minuto, por causa de um problema dos Correios, teve que ir pessoalmente pegar minha documentação nos Correios e entregar na coordenadoria da Pós de Computação. Sem ela eu não teria nem participado do processo seletivo do mestrado.

À Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pela concessão de bolsa de estudos para cursar o mestrado durante o segundo semestre de 2017 até Fevereiro de 2019. Sem a bolsa, seria praticamente impossível ter concluído esse curso.

E por fim, à Força que esteve comigo sempre.

Resumo

Desenvolver agentes artificiais capazes de jogar bem é um objetivo de longa data na área de IA para jogos. Recentemente, surgiram vários casos de sucesso, impulsionados principalmente por técnicas que reúnem Aprendizagem por Reforço (RL) e aprendizagem baseada em redes neurais. Entretanto, a maioria dos resultados foi obtido usando apenas as imagens gráficas da tela do jogo, com recursos físicos caros para se efetuar os cálculos e com importantes restrições pela falta de conhecimento do contexto do jogo. Nesta dissertação propomos o desenvolvimento de um Ambiente de Aprendizado aplicado ao jogo Bomberman (BLE) com o objetivo de verificar se os algoritmos de RL conseguem aprender a jogar sem informações privilegiadas. Escolhemos o jogo Bomberman por ser um jogo com regras claras e ser um jogo amplamente adotado como plataforma de testes na literatura do assunto. O BLE inclui o algoritmo de Otimização de Política Proximal (PPO) que pode ser usado com um Perceptron de Múltiplas Camadas e/ou com uma Rede de Memória de Curto Longo Prazo (LSTM), a possibilidade de treinamento por imitação, seguido ou não por PPO, e a opção de aprender com a experiência de um ou vários agentes de uma só vez. Além disso, o BLE fornece cinco maneiras diferentes de representar o espaço de estados. Foram realizadas várias experiências de treinamentos e torneios em que os agentes jogam uns contra os outros para selecionar a melhor representação e técnica de aprendizado nesse cenário. Os resultados apontaram que, usando uma representação híbrida, agregando as experiências dos vários agentes para aprender de uma só vez e utilizando PPO com LSTM, são produzidos agentes inteligentes do Bomberman que vencem mais partidas entre todos os agentes treinados.

Palavras-chave: bomberman, aprendizado por reforço, aprendizagem por imitação, redes neurais, representação de estado, unity3d, ml-agent toolkit, proximal policy optimization, lstm.

Abstract

Making artificial agents capable of playing well games is a long-standing goal in the area of Game AI. Recently, a number of successful cases have emerged driven mainly by techniques that put together Reinforcement Learning (RL) and neural network-based learning. However, in most of the cases, the results have been achieved by training directly from pixels with expensive resources. On the other hand, a universally pleasurable game that requires agents learning from complex environments but still not necessarily from the pixels is Bomberman. In this research, we devised a Bomberman Learning Environment (BLE) aiming at testing whether RL algorithms have the ability to learn how to play Bomberman without privileged information. BLE includes the Proximal-Policy Optimization (PPO) algorithm that can be used with a Multi-Layer Perceptron and/or a Long-Short Term-Memory network (LSTM), the possibility of training with imitation learning followed or not by PPO, and the option of learning from the experience of one or several agents at once. In addition, BLE provides five different ways of representing the space of states. We conducted several pieces of training and tournament experiments where the agents play against each other to select the best representation and learning technique in this scenario. The results have pointed out, in most cases, that by using a hybrid representation, aggregating the experiences of the several agents to learn at once, and using PPO with LSTM we have Bomberman agents that know how to win the game against the other trained agents.

Keywords: bomberman, reinforcement learning, imitation learning, neural networks, state representation, unity3d, ml-agent toolkit, proximal policy optimization, lstm.

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

A3C	:	actor-critic (ator-crítico);
AI	:	Artificial Intelligence (Inteligência Artificial);
BC	:	Behavioral Cloning (Clonagem Comportamental);
CNN	:	Convolutional Neural network (Redes Neurais Convolucionais);
DL	:	Deep Learning (Aprendizagem Profunda);
DQN	:	Deep Q-Network (Rede-Q Profunda);
DRL	:	Deep Reinforcement Learning (Aprendizagem por Reforço em Redes Profundas);
GRU	:	Gated Recurrent Unit (Unidade Recorrente Bloqueada);
IDE	:	Integrated Development Environment (Ambiente de Desenvolvimento Integrado);
IL	:	Imitation Learning (Aprendizagem por Imitação);
IRL	:	Inverse Reinforcement Learning (Aprendizagem por Reforço Inversa);
LSTM	:	Long Short Term Memory (Redes de Memória de Longo-curto Prazo);
MDP	:	Markov Decision Process (Processo de Decisão de Markov);
ML	:	Machine Learning (Aprendizagem de Máquina);
MLP	:	Multilayer Perceptron (Perceptron Multicamadas);
ML-Agents	:	Unity Machine Learning Agents Toolkit;
NN	:	Neural Networks (Redes Neurais);
POMDP	:	Partially Observable Markov Decision Process; (Processo de Decisão de Markov Parcialmente Observável)
PPO	:	Proximal Policy Optimization (Otimização de Política Proximal);
RL	:	Reinforcement Learning (Aprendizagem por Reforço);
RNN	:	Recurrent Neural Network (Redes Neurais Recorrentes);

Sumário

Capítulo 1

Introdução

Neste capítulo inicial, apresentamos o contexto e a motivação pelos quais esta dissertação foi produzida, dando uma visão geral do que é o jogo Bomberman e a importância desse jogo como ambiente para o desenvolvimento de novos algoritmos e métodos na área da Inteligência Artificial (AI). Além disso, informamos brevemente como trabalhos relacionados abordaram o problema de resolver o Bomberman. Por último, apresentamos os problemas abordados neste trabalho, nossa metodologia e nossas contribuições. Tornamos a nossa solução publicamente disponível num repositório[1], permitindo futuras melhorias e uso adicionais.

1.1 Motivação

Produzir jogos com entidades que exibem inteligência similar ao nível humano ou sobre humana é um objetivo de longa data na área de Inteligência Artificial (AI) aplicada a jogos (*Game AI*) [22, 20, 49]. Esse é um objetivo almejado em diversas áreas da AI, tais como, Busca, Planejamento, Tomada de Decisões, e, em especial, Aprendizagem de Máquina [30, 47, 49, 48, 35, 36, 6], particularmente aqueles desenvolvidos na área de Aprendizagem por Reforço (RL)[51]. Recentemente, nos estudos de AI, algoritmos de RL via redes neurais (profundas) (DRL) [25] mais uma vez atraíram a atenção de pesquisadores, profissionais e da indústria de jogos e entretenimento digital. Um dos motivos que despertaram essa atenção foi a vitória por 4 a 1 do sistema de AI AlphaGo [47], desenvolvido pela empresa Google DeepMind, sobre o sul-coreano, Lee Sedol, 18 vezes campeão mundial do jogo de tabuleiro Go[2]. Além disso, a Google DeepMind desenvolveu outra versão do método de

[1] Código fonte, vídeos e outras informações no link https://github.com/lorel-uff/pip
[2] https://en.wikipedia.org/wiki/AlphaGo_versus_Lee_Sedol

AI que joga Go (AlphaGo Zero), porém, que aprende jogando sozinho em partidas contra ele mesma [49]. Após treinado, ele venceu de 100 a 0 o AlphaGo. O Google DeepMind ainda desenvolveu uma nova versão chamada Alpha Zero [48], que possui um método mais genérico e consegue jogar vários jogos de tabuleiro, entre eles xadrez, Shogi e go, com o mesmo algoritmo e a mesma arquitetura de rede neural. A critério de comparação, o Alpha Zero jogando com as peças brancas venceu $68,9\%$ das partidas contra o AlphaGo Zero. Com as peças pretas, o Alpha Zero venceu $53,7\%$ das partidas.

Geralmente, os agentes inteligentes mais famosos são treinados para vencer jogadores humanos em jogos de tabuleiro, como no caso dos algoritmos AlphaGo (Go), AlphaGo Zero (Go), Alpha Zero (Go, Shogi e Xadrez) e Deep Blue (Xadrez) [15]. Na área dos jogos digitais, houve também um avanço no desenvolvimento de agentes com DRL para jogar jogos de Atari [30]. Frames capturados dos jogos são usados como observação para alimentar a rede neural. Recentemente, a empresa OpenAI desenvolveu e está desenvolvendo agentes capazes de jogar um MOBA[3] chamado Dota2 [36, 35]. MOBA é um jogo complexo e caso eles sejam capazes de criar uma AI que vença jogadores profissionais no modo 5×5 será um grande avanço nas pesquisas de AI para jogos. Outro caso de sucesso, foi o sistema de AI AlphaStar que aprendeu a vencer humanos no jogo Starcraft 2 [6]. Todavia, o AlphaStar, diferentemente dos jogadores profissionais humanos, pode saber o que está acontecendo em todos os locais do mapa ao mesmo tempo, sem precisar mover o mapa.

Na maioria dos casos citados anteriormente são necessários recursos computacionais caros para lidar com um enorme espaço de pesquisa. Escolhemos Bomberman [5], por ser um jogo de estratégia baseado em labirinto, universalmente agradável e famoso, e que ainda requer inteligência para ser jogado, mas pode ser representado por um cenário de grade simples. Embora à primeira vista pareça simples, projetar um agente inteligente para jogar o Bomberman requer lidar com uma série de desafios, como um vasto espaço de busca devido a inúmeras possibilidades de componentes (outros agentes, bombas, blocos, *etc*), modo multijogador, raciocínio estratégico, recompensa atrasada por causa do tempo que as bombas levam para explodir, e um ambiente dinâmico. Por causa de tais características, Bomberman é considerado um cenário atraente para o desenvolvimento de métodos de AI [41].

A maioria dos trabalhos disponíveis que trata de Bomberman concentrou-se no desenvolvimento de agentes reflexivos, ou através de técnicas de busca e planejamento, ou

[3]https://pt.wikipedia.org/wiki/Multiplayer_online_battle_arena

usando RL baseado em tabelas, sem explorar o poder de aproximação de funções mais gerais com redes neurais [5, 28, 7, 31]. Recentemente, Komerlink *et al.* [21] usou o Q-learning acoplado a uma rede neural perceptron multicamada (MLP), focando na comparação de estratégias de exploração seguidas pelo algoritmo Q-learning. Os trabalhos [5, 28, 7] desenvolveram agentes de RL que não utilizavam redes neurais. Contudo, estes agentes apenas conseguiram aprender a jogar versões simplificadas do jogo Bomberman por causa das limitações do algoritmo de RL tabulares, tais como o tempo necessário para percorrer todas as células da tabela usada para armazenar o valor de cada estado, como também o espaço de armazenamento utilizado para salvar esta tabela. Monteiro *et al.* [31] criou um agente que utiliza algoritmos de busca por caminhos (BFS e A*) e árvore de decisão, e este agente conseguiu completar as fases do primeiro jogo original do Bomberman. Já em [5, 41] o foco foi em desenvolver ambientes de aprendizagem para a criação de agentes Bomberman, e o [41] atualmente está sendo usado em algumas competições, em que agentes que utilizam novos métodos estão sendo testados. Em resumo, os trabalhos relacionados não conseguem produzir um agente inteligente que utilize apenas algum método de RL para jogar uma versão do Bomberman que tenha bombas, blocos destrutíveis e inimigos dinâmicos. Mais informações sobre os trabalhos relacionados podem ser encontradas no Capítulo 3.

Nesta dissertação, incrementamos os ambientes de aprendizagem desenvolvidos anteriormente com: (1) a capacidade de representar células com mais de um estado, (2) a habilidade de ter agentes que possam ocupar a mesma célula como no jogo original, (3) um ambiente dentro de um motor de jogo com uma vasta comunidade de desenvolvedores (Unity3d) [54]. Percebemos que desenvolver agentes com apenas aprendizagem por reforço ou por imitação sem utilizar informações privilegiadas (heurísticas, busca e planejamento e árvores de decisão) que consigam jogar o jogo Bomberman contendo suas características mais básicas tais como aprender a sobreviver, colocar bombas e fugir delas, destruir blocos e matar inimigos é um problema não solucionado nos trabalhos relacionados.

1.2 Objetivos

O objetivo principal desta dissertação é obter a partir da aprendizagem por reforço e por imitação que utilizam redes neurais, um agente que consiga jogar o Bomberman. Por meio de uma série de treinamentos e torneios que realizamos, pretendemos descobrir a melhor forma de representarmos o estado das células da grade do cenário do jogo, encontrar o melhor algoritmo de aprendizagem disponível no nosso ambiente de aprendizagem

de Bomberman (intitulado como BLE), decidir se é melhor que os agentes sejam treinados em conjunto ou isoladamente, e conseguir efetivamente continuar um treinamento com o algoritmo de Otimização de Política Proximal (PPO) inicializando seus neurônios (tensores) com valores carregados de um modelo gerado num treinamento prévio com o algoritmo de Clonagem Comportamental (BC).

Nossa hipótese é que algoritmos de aprendizagem por observação baseados em otimização de política produzem agentes capazes de jogar um jogo que tenha características tais como recompensa atrasada, ambientes dinâmicos, e cenários que possam ser representados por meio de uma grade (grid) que podem conter ou não entidades (e.g. agente, bomba, bloco, entre outras) em suas células. Para tanto, precisamos representar o espaço de estados de modo que o algoritmo de aprendizagem consiga aprender os padrões do jogo, visto que a aprendizagem por observação se baseia em um agente que observa e age no ambiente. Em seguida precisamos descobrir e selecionar o melhor algoritmo de observação entre os presentes no BLE: aprendizagem por reforço via redes neurais especificamente PPO e PPO com a rede recorrente Long Short-Term Memory (LSTM), e aprendizagem por imitação utilizando BC, sozinha ou seguida de um treinamento posterior com o PPO. Nossa ênfase é que treinar previamente um agente com BC alavancará um treinamento posterior com o PPO. Para testar esta hipótese, nós incluímos os componentes acima em um ambiente de aprendizagem e treinamento. Desenvolvemos o BLE para confirmar nossa hipótese e permitir a execução de treinamentos e torneios. Com isso, através destes torneios, nós conseguimos encontrar os métodos e representações mais vitoriosos.

1.3 Metodologia

Nesta dissertação seguimos um caminho diferente dos trabalhos existentes na literatura principalmente em relação ao método de aprendizado: a complexidade do jogo Bomberman foi abordada com o poder de expressividade da recém-desenvolvida estratégia de Otimização de Políticas Proximal (PPO) [45] que utiliza o framework *actor-critic* [29], desenvolvido dentro da área de DRL. Além disso, permitimos a integração do PPO com a Aprendizagem por Imitação (IL) [25], onde um agente aprendiz (aluno) aprende a executar uma tarefa observando um especialista. Discutivelmente, essa estratégia pode produzir um bom ponto de partida para o algoritmo PPO. Desenvolvemos o *Bomberman Learning Environment* (BLE), um ambiente de aprendizagem do Bomberman que pode ser utilizado para testar diferentes algoritmos de aprendizado e também para permitir diferentes formas de representação de estado, já que este é um componente essencial do processo de

decisão de Markov subjacente aos algoritmos de RL.

O jogo Bomberman desenvolvido no BLE consegue representar células com mais de um elemento, permitindo que mais de um agente ocupe a mesma célula, como no jogo original. Além disso, os agentes incluídos no BLE não recebem informações privilegiadas sobre o ambiente do jogo, pois, o algoritmo de RL utilizado não é integrado nem com busca e planejamento do espaço de estados, nem com nenhuma heurística. Dessa forma, pode-se avaliar a capacidade das estratégias de DRL para aprender a jogar Bomberman com o mínimo de informações do ambiente. O BLE inclui cinco formas de se representar o espaço de estados da grade do jogo: Flag Binária, Flag Binária Normalizada, ICAART, Híbrida e ZeroOuUm, e possui os seguintes algoritmos de treinamento: PPO com MLP, PPO com redes de memória de curto longo prazo [14] (LSTM), o algoritmo de IL chamado Clonagem Comportamental (BC), BC seguido por PPO. O BLE também permite treinamentos que agregam a experiência de vários agentes de uma só vez para produzir os exemplos para aprendizagem da função de aproximação, ou treinar modelos concomitantemente, que são oponentes e agregam somente a experiência de agentes específicos durante o treinamento. A implementação dos algoritmos de treinamento se deu no ML-Agents Toolkit [18], um projeto de código aberto para pesquisa em AI e para jogos.

1.4 Contribuições

Apresentamos primeiramente, as contribuições mais importantes produzidas nesta dissertação:

1) Desenvolvimento de novas formas de representação do estado para o jogo Bomberman: desenvolvemos 4 novas formas de representação do estado (Híbrida, ZeroOuUm, Flag Binária, Flag Binária Normalizada) e implementar a representação ICAART desenvolvida em [21]. Ademais, fizemos um estudo comparativo entre estas formas de se representar o espaço de estados de um jogo estilo Bomberman. Descobrimos que a representação Híbrida, dentre as representações analisadas, é a melhor forma.

2) Desenvolvimento de um novo método em que o agente usa o treino prévio com o BC para iniciar o treinamento posterior com o PPO: nos experimentos, o treinamento prévio com BC influenciou positivamente um treinamento posterior com o PPO. Acreditamos que métodos como esse possuem um grande potencial para alavancar o treinamento de agentes inteligentes em jogos e aplicações, porque um agente aprendiz por meio da imitação consegue aprender sequências de ações complexas num num

curto intervalo de tempo, e através de um treinamento continuado com o PPO, o agente consegue complementar seu treinamento por meio da tentativa e erro, consequentemente aprimorando a política aprendida no treinamento anterior além de aprender a escolher ações quando está num estado que não foi visto no treinamento prévio. Nessa abordagem, o agente começa a aprender a imitar o jogador utilizando os dados de repetição do jogo para alavancar o tempo de treinamento. Em seguida, após a imitação bem sucedida, o agente continua o aprendizado sozinho via RL com o PPO. Caso o treinamento não for complementado com o PPO, o agente só sabe escolher uma ação coerente caso a observação do estado atual do jogo já tiver sido vista em seu treinamento com o BC. Portanto, continuar o treinamento com PPO é fundamental para que o agente consiga generalizar sua política aprendida.

A seguir, apresentamos as contribuições secundárias que também foram produzidas nesta dissertação.

i): Desenvolvemos um novo ambiente de aprendizagem chamado BLE. Ele pode ser compartilhado com outros pesquisadores e estudantes e se tornar um ambiente público para se testar e criar métodos e algoritmos. Além disso, ele pode ser estendido com a criação de novas regras, poderes, recompensas, observações, tamanhos de cenários diversificados, etc. Também há um ambiente simplificado do jogo Bomberman, chamado de Hammerman (ver Apêndice B) onde não há bombas e o objetivo do agente é alcançar algum objetivo no cenário, abrindo caminho com o uso de seu martelo "imaginário".

ii): Realizamos a modelagem do jogo Bomberman como um Processo de Decisão de Markov (MDP), definindo os estados, ações, recompensas e transições. As recompensas foram definidas, ajustadas e calibradas para fazer com que o agente conseguisse aprender com o método PPO.

iii): Desenvolvemos um agente inteligente que aprendeu a jogar Bomberman utilizando o algoritmo PPO, com LSTM ativado ou não. Além disso, por meio do algoritmo BC o agente aprendeu a copiar os movimentos demonstrados por um especialista.

iv): Compararamos o treinamento entre Brains do kit de ferramentas ML-Agents (a seção 2.3 explica o que este kit), quando utilizado um treinamento Multi-Brain ou individual. Foi descoberto que apesar de o treinamento Multi-Brain obter melhores estatísticas na fase de treinamento, esse modo de treino é inferior ao treinamento de apenas um Brain por vez. Note que vários agentes podem estar atrelados a um único Brain. Um Brain é uma entidade do ML-Agents responsável por aprender e fornecer uma política de tomada de decisão para cada um dos agentes associados a ele.

Os resultados obtidos nesse trabalho apontam que usando a representação de estado proposta neste trabalho e agregando as experiências de vários agentes em um treinamento com PPO com LSTM é a melhor maneira de conseguir um agente Bomberman eficaz contra oponentes variados.

Também é importante ressaltar o que não é escopo deste trabalho: tratar espaço e tempo não discretizados, habilidades extras (power-ups), modo multijogador em times, implementação de algoritmos diferentes do PPO, LSTM e BC, e alimentar os algoritmos com observações visuais tais como imagem de câmeras, texturas e etc.

1.5 Organização da Dissertação

Esta dissertação está organizada da seguinte forma. A fundamentação teórica é explicada no Capítulo 2 e os trabalhos relacionados no Capítulo 3. Depois, apresentamos o BLE no Capítulo 4, seguido dos resultados experimentais e análises no Capítulo 5, e finalizamos o trabalho com as conclusões e trabalhos futuros no Capítulo 6.

Capítulo 2

Fundamentação Teórica

Neste capítulo apresentamos os conceitos preliminares e fundamentais para que todo assunto abordado nesta dissertação seja corretamente compreendido. Modelamos o BLE como um Processo de Decisão de Markov (MDP), contendo um conjunto de estados, ações, recompensas e transições; o conceito do MDP é explicado na Seção 2.1. Utilizamos um algoritmo de Aprendizagem por Reforço (Seção 2.2) que utiliza redes neurais artificiais (Subseção 2.2.2) para aprender por meio de aproximação de função, uma política para jogar o Bomberman. O algoritmo de Aprendizagem por Reforço via Aprendizagem Profunda utilizado é o algoritmo de Otimização de Política Proximal (Subseção 2.2.4) que pode ser usado em conjunto com redes de memória de longo-curto prazo (Subseção 2.2.3). Também explicamos a Aprendizagem por Imitação que foi usada em alguns experimentos na Subseção 2.2.5. Por fim, o funcionamento básico do kit de ferramentas ML-Agents que foi amplamente utilizado nesta pesquisa é explicado na Seção 2.3, e uma introdução a dinâmica do jogo Bomberman original é dada na Seção 2.4.

2.1 Processos de Decisão de Markov

Markov Decision Process (MDP), em português Processo de Decisão de Markov, é uma formalização clássica da tomada de decisões sequenciais para um ambiente estocástico totalmente observável com um modelo de transição Markoviano (probabilidade de alcançar s^l de s depende apenas de s e não do histórico de estados anteriores) e recompensas aditivas [42]. Num MDP, as ações influenciam não apenas as recompensas imediatas, como também os estados subsequentes e as futuras recompensas. Assim, os MDPs envolvem recompensas atrasadas e a necessidade de compensar recompensas imediatas e atrasadas [51]. MDPs representam o modelo do mundo através de estados, ações, modelo de transi-

ção e recompensas. São chamados *de Markov* porque os processos modelados obedecem à propriedade de Markov: que o estado atual depende apenas de um número fixo finito de estados anteriores, e o tipo de MDP mais simples é o MDP de primeira ordem, em que o estado atual depende apenas do estado anterior e não de todos os estados anteriores[42]; e são chamados de *processos de decisão* porque modelam a possibilidade de um agente interferir periodicamente no sistema executando ações, diferentemente de Cadeias de Markov, onde não se trata de como interferir no processo.

Como as decisões dos agentes devem ser tomadas em sequência e a consequência de cada decisão não está clara para o agente, formulamos estas situações como MDPs. Pellegrini et al. diz que o MDP é uma tupla (S, A, T, R) [39] onde:

- S é o conjunto de estados do ambiente em que o agente pode estar.
- A é um conjunto de ações que podem ser executadas em diferentes épocas de decisão. Há a possibilidade também de existirem ações possíveis específicas para cada estado.
- $T : S \times A \times S \mapsto [0, 1]$ é uma função que dá a probabilidade do agente passar para um estado $s^{'} \in S$, dado que o agente estava em um estado $s \in S$ e o agente decidiu executar uma ação $a \in A$ (denotada $T(s^{'}|s, a)$).
- $R : S \mapsto \mathbb{R}$ é uma função que dá a recompensa para o agente quando ele está em um estado $s \in S$.

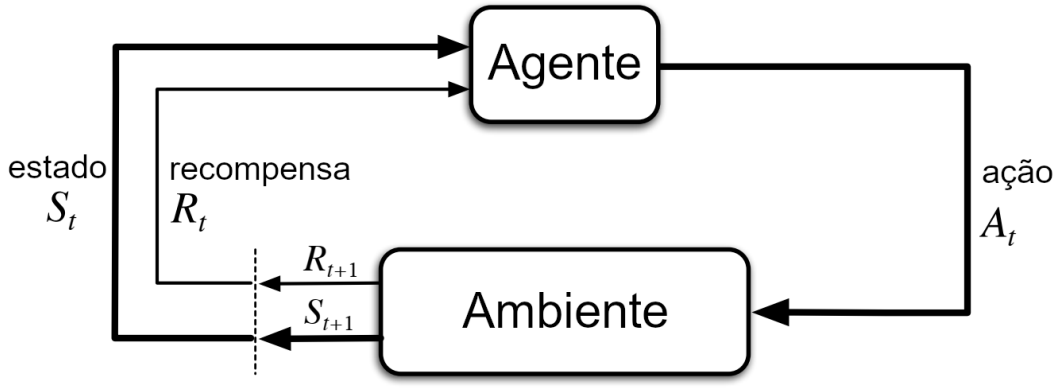

Figura 2.1: A interação do agente com o ambiente em um MDP. (Fig. traduzida de [51])

Na Figura 2.1 é mostrado o fluxo da interação de um agente com um ambiente onde a interação é modelada como um MDP. No estado corrente, o agente age no ambiente e este

retorna um novo estado e uma recompensa para o agente. Este laço se repete arbitrariamente. Para se resolver um problema MDP torna-se necessário mapear uma ação para cada estado do ambiente. Este mapeamento é chamado de política (π), ou seja, a política informa o que o agente deve fazer em cada estado do ambiente. A qualidade de uma política é medida pela utilidade esperada (U_h) da sequência de ações geradas, computada a partir das recompensas obtidas em cada estado. A função de utilidade de uma política pode ser aditiva ou com desconto (Veja as Equações 2.1 e 2.2, respectivamente). O fator de desconto é representado por γ e possui valores entre [0, 1]. Este fator determina o valor presente das recompensas futuras: uma recompensa recebida em intervalos de tempo no futuro vale apenas γ^{k-1} vezes o que valeria a pena se fosse recebida imediatamente [51].

$$U([s_0, s_1, s_2, ...]) = R(s_0) + R(s_1) + R(s_2) + ... \tag{2.1}$$

$$U([s_0, s_1, s_2, ...]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + ... \tag{2.2}$$

Por fim, uma política ótima é representada por π^*. $U^\pi(s)$ é a utilidade esperada usando a política π no estado s. De forma geral, uma π^* deve satisfazer a seguinte equação:

$$U^\pi(s) = E[\sum_{t=0}^{\infty} \gamma^t R(s_t)]$$
$$\arg\max_\pi U^\pi(s) \rightarrow \pi^*(s) = \arg\max_{a \in A(s)} \sum_{s'} P(s'|s,a)U(s') \tag{2.3}$$

Quando se possui o modelo do mundo, os principais algoritmos que encontram a política ótima são os baseados em *Value Iteration* ou em *Policy Iteration*. Métodos baseado em *Value Iteration* tem como ideia básica computar a utilidade de cada estado e usá-la para selecionar a melhor ação. A utilidade de um estado é calculada usando a fórmula da Equação 2.2. O cálculo de $U(s)$ permite selecionar ações utilizando o princípio da Utilidade Máxima Esperada: basta escolher a ação que maximiza a utilidade esperada do próximo estado [42]

$$\pi^* = \arg\max_a \sum_{s'} T(s, a, s')U(s') \tag{2.4}$$

Podemos escolher a ação no estado atual que maximiza a utilidade do próximo estado. Deste modo, escolhemos a ação, analisando a utilidade esperada dos estados vizinhos do estado corrente. A utilidade de um estado é a recompensa imediata mais a utilidade (com desconto) esperada daquele estado. A Equação 2.5 é conhecida como Equação de

Bellman.

$$U(s) = R(s) + \gamma \max_{a} \sum_{s'} T(s, a, s')U(s') \tag{2.5}$$

Em resumo, o *Value Iteration* é um algoritmo que atribui um valor arbitrário para as utilidades iniciais e faz várias iterações da equação de Bellman, até que o equilíbrio seja atingido. Já o algoritmo *Policy Iteration*, diferente do algoritmo anterior, usa a equação de Bellman sem o operador max, inicia o algoritmo com uma política aleatória, configura o valor da política atual e depois, de maneira gulosa, aperfeiçoa a política, buscando modificar as ações recomendadas para cada estado.

2.2 Aprendizagem por Reforço

Na Aprendizagem por Reforço (RL), o agente aprende por tentativa e erro, mapeando situações em ações (política) e recebendo recompensas positivas quando escolhe ações corretas e recompensas negativas quando escolhe ações incorretas, ou seja, o agente busca maximizar um sinal de recompensa numérico. Duas características distintivas importantes em RL são a busca por tentativa e erro, e a recompensa atrasada [51].

A RL utiliza-se da estrutura formal dos MDPs para definir a interação entre um agente aprendiz e seu ambiente em termos de estados, ações e recompensas. Esta estrutura formal tem o objetivo de ser uma forma simples que representa as características mais importantes do problema de AI. Essas características incluem um senso de causa e efeito, um sentimento de incerteza e não-determinismo e a existência de objetivos explícitos [51].

A RL pode ser vista como um terceiro paradigma de ML, visto que não é aprendizado supervisionado e nem aprendizado não supervisionado. Aprendizado supervisionado é o aprendizado de conjuntos de treinamento de exemplos já classificados por um especialista externo. O objetivo do treinamento supervisionado é aprender a generalizar novas classificações por meio desse treinamento prévio com exemplos já rotulados. Aprendizado não supervisionado consiste em encontrar estruturas escondidas ou novos padrões em dados não rotulados. A RL tenta maximizar um sinal de recompensa em vez de tentar encontrar uma estrutura oculta.

O dilema entre explorar ou usufruir é outro desafio que surge na RL e não em outros tipos de aprendizagem. O agente deve escolher uma ação que lhe traz uma recompensa já conhecida ou deve explorar o ambiente escolhendo outra ação que pode ser melhor ou pior que a ação já conhecida? Em resumo o dilema é que nem a exploração, nem o usufruto

de ações já conhecidas podem ser realizadas exclusivamente, sem que o agente falhe na tarefa. O agente deve tentar executar uma variedade de ações e favorecer progressivamente aquelas que parecem ser as melhores.

Alguns problemas que a RL possuem durante o treinamento são: o atraso na convergência (muito tempo necessário para se aprender tarefas complexas), o alto custo computacional (quando se utiliza imagens como observação o custo computacional é maior ainda), o grande volume de dados (o agente precisa ter muitas experiências para aprender com eficiência uma tarefa), torna-se necessário existir alguma função de recompensa. Além disso, a RL é muito sensível tanto à sua inicialização quanto à dinâmica de seu processo de treinamento, porque seus dados são sempre coletados on-line, ou seja, geralmente o treinamento precisa ser feito num ambiente simulado (e.g. antes de treinar com um carro autônomo real, o agente de RL deve treinar num ambiente simulado). Para mais, a única supervisão que um agente de RL recebe é uma recompensa, muitas vezes representada por um simples valor numérico, e modelar boas funções de recompensa é uma tarefa difícil [17], e.g. no jogo CoastRunner 7 da OpenAI, a má modelagem da função de recompensa fez o barco procurar por apenas itens na fase, em vez de se preocupar em vencer a corrida [4].

2.2.1 Conceitos fundamentais de RL

Num sistema de RL é possível identificar seis elementos principais: o agente, o ambiente, a política, o sinal de recompensa, a função de valor e, opcionalmente, um modelo do ambiente. A política pode ser entendida como um mapeamento de estados percebidos do ambiente para ações a serem tomadas quando se está em tais estados. O sinal de recompensa é um número enviado pelo ambiente, a cada etapa de tempo (quando dado apenas no final de um episódio, a recompensa se torna esparsa), ao agente. Este sinal pode ser positivo ou negativo. Com isso, o objetivo do agente é maximizar a recompensa total que ele recebe ao longo do tempo. O principal fator que pode alterar a política é o sinal de recompensa; se uma ação selecionada pela política for seguida de baixa recompensa, a política poderá ser alterada para selecionar alguma outra ação nessa situação no futuro. Resumidamente, a função de valor de um estado é o valor total de recompensa que um agente pode esperar acumular no futuro, a partir desse estado, ou seja, a função de valor tem a mesma definição que a utilidade esperada do MDP. Enquanto o sinal de recompensa é um valor imediato recebido num estado, a função de valor leva em conta o valor que o estado atual pode representar no futuro. Por exemplo, um estado pode sempre render

uma baixa recompensa imediata, mas ainda assim ter uma função de valor alta, porque é regularmente seguido por outros estados que geram altas recompensas. Sem recompensas não poderia haver função de valor, e o único propósito de estimar valores é para conseguir mais recompensa. Por fim, o modelo do ambiente é o que permite inferências sobre como o ambiente se comportará.

Os métodos para resolver problemas de RL que usam modelos e planejamento são chamados de métodos baseados em modelo, em oposição a métodos mais simples, livres de modelos que explicitamente aprendem por tentativa e erro, e que são vistos como quase opostos do planejamento. Outra definição importante que precisamos saber é a diferença entre métodos *on-policy* e *off-policy*. Os métodos *on-policy* tentam avaliar ou melhorar a política usada para tomar decisões, enquanto os métodos *off-policy* avaliam ou melhoram uma política diferente daquela usada para gerar os dados [51]. De forma mais detalhada, um método *off-policy* utiliza duas políticas, uma que é aprendida e que se torna a política ideal, e uma que é mais exploratória e é usada para gerar comportamento. A política que está sendo aprendida é chamada de política alvo, e a política usada para gerar comportamento é chamada de política de comportamento. Neste caso, dizemos que a aprendizagem é a partir de dados "off"da política alvo, e o processo global é denominado aprendizagem *off-policy* [51].

Como foi dito na Seção 2.1, há algoritmos que conseguem encontrar a política ótima, diretamente ou indiretamente (utilizando os valores dos estados calculados). Porém, nem todos os problemas possuem o espaço de estado de tamanho limitado o suficiente ao ponto de permitirem calcular a melhor política para cada estado. Dependendo do problema a ser solucionado, o custo computacional pode ser extremo por causa do vasto espaço de busca num conjunto de estados extensos. Além disso, outro fator limitante quanto ao cálculo da política é a quantidade de memória que o computador possui para armazenar cada função de valor de cada estado. Geralmente, os algoritmos de RL utilizam tabelas para armazenar estes valores das funções de valor, e são conhecidos como algoritmos de RL tabulares (e.g. Q-Learning utiliza tabelas pra armzazenas funções de valor de pares estado/ação). Contudo, armazenar os resultados em tabelas é limitante, porque elas podem ocupar muito espaço na memória do computador e consequentemente causar um vazamento de memória, além disso, iterar sobre o conjunto de estados para se atualizar a política e os próprios valores de cada estado pode levar um tempo que tende ao infinito. Em resumo, não há tempo para explorar todo conjunto de estados. Por conta disso, existem métodos de aproximação de funções que substituem a necessidade de se utilizar tabelas. Como diz [25], a aproximação de função é uma forma de generalização quando o estado e/ou

espaços de ação são grandes, ou contínuos, visando generalizar uma aproximação de toda a função a partir de exemplos dela. Há aproximações de função lineares e não lineares. Nesta dissertação, utilizamos apenas aproximações não lineares.

Por conta da complexidade do número de estados e/ou ações que um ambiente pode ter, geralmente usam-se funções não lineares de representação que mapeiam ambos estados e ações em valores. Em ML isto é chamado de "problema de regressão" [24]. Este problema pode ser solucionado de diferentes formas, porém, umas das formas mais eficazes é usar redes neurais para aprender a calcular a política. Em resumo, o objetivo é tentar aproximar uma função não linear complexa com uma rede neural. As redes neurais podem focar em aprender a otimizar diretamente as políticas (métodos baseados em política) ou focar em aprender as funções de valor primeiro para depois, através destas, inferir as políticas (indiretamente calcula a política). O método DQN [30], famoso por conseguir fazer um agente jogar jogos do videogame Atari [30], por exemplo, calcula a política indiretamente.

Os algoritmos de RL, que se aproveitam de redes neurais, desenvolvidos na atualidade, geralmente, buscam encontrar a política diretamente ao invés de indiretamente (utilizando-se dos maiores valores calculados através de funções de valor). Ou seja, esses algoritmos de RL que usam redes neurais aproximam funções usando gradiente de política. O gradiente de política define a direção na qual é necessário alterar os parâmetros da rede para melhorar a política em termos da recompensa total acumulada, que significa que o objetivo é tentar aumentar a probabilidade de ações que dão uma recompensa total satisfatória e diminuir a probabilidade de ações com resultados finais ruins [24].

2.2.2 Aprendizagem via Redes Neurais

As redes neurais artificiais (NNs) consistem em muitos processadores simples e conectados, chamados neurônios, cada um produzindo uma sequência de ativações de valor real. Os neurônios de entrada são ativados através de sensores que percebem o ambiente, outros neurônios são ativados através de conexões ponderadas de neurônios previamente ativos. Alguns neurônios podem influenciar o ambiente ao desencadear ações. Já Aprendizagem via NNs é sobre encontrar pesos que fazem a rede exibir o comportamento desejado [43]. A Figura 2.2) exibe um exemplo de rede neural artificial contendo 3 neurônios na camada de entrada, 4 na primeira camada escondida, 2 na segunda camada escondida, e 2 na camada de saída.

Deep Learning (DL) [9], em português, Aprendizagem Profunda, é uma subcategoria

de ML que utiliza redes neurais artificiais para modelar abstrações de alto nível de dados usando um grafo profundo com diversas camadas escondidas entre a camada de entrada e a camada de saída, que com o treinamento, aprenderá a reconhecer padrões. A DL consegue aprender representações de forma automática a partir de entradas brutas para recuperar as hierarquias de composição de muitos sinais naturais [25]. Atualmente há um debate na literatura discutindo até que ponto uma NN pode ser considerada profunda ou não [33]. Nesta dissertação, como não utilizamos muitas camadas ocultas, consideramos que nossa NN não é profunda.

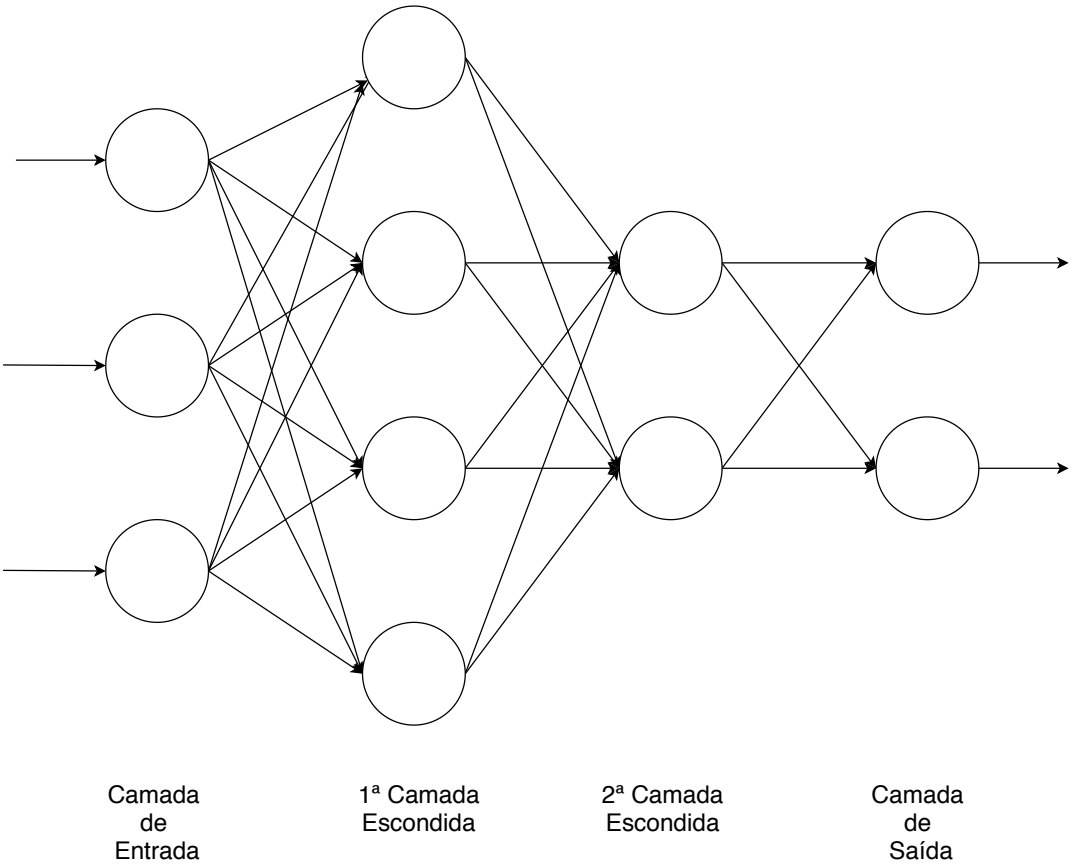

Figura 2.2: Exemplo de Rede Neural Artificial (Fig. inspirada na Fig. de [23])

2.2.2.1 Noções Básicas de uma rede Neural Artificial

Em NN, com exceção da camada de entrada, em cada camada, é calculada a entrada para cada unidade (neurônio). Este cálculo é a soma ponderada das unidades da camada anterior. Então, tipicamente usa-se uma transformação não-linear ou uma função de ativação para aplicar à entrada de uma unidade, com o intuito de obter uma nova representação da entrada da camada anterior. Há pesos nos links entre as unidades de camada para camada. Depois que os cálculos fluem para frente (da entrada para a saída), na camada

de saída e em cada camada escondida, pode-se calcular derivadas de erro para as camadas anteriores, e consequentemente pode-se calcular gradientes de retro-propagação para a camada de entrada, para que os pesos possam ser atualizados visando otimizar alguma função de perda [25].

Há três tipos de redes neurais artificiais bem conhecidas: perceptron multicamadas (MLP), convolucionais (CNN), e recorrentes (RNN). MLPs são redes que mapeiam um conjunto de valores de entrada em valores de saída com uma função matemática formada pela composição de muitas funções mais simples em cada camada. CNNs são redes MLP com camadas convolucionais, camadas de agrupamento e camadas totalmente conectadas. RNNs são redes que possuem recorrência e por causa disto, é esperado que o histórico e elementos passados possam ser armazenados para reforçar experiências boas ou ruins, porém, elas não conseguem armazenar informações por muito tempo (Veja a Figura 2.3). As redes de memória de longo-curto prazo (LSTM) e unidades recorrentes bloqueadas (GRU) [3] foram propostas para tratar de tais questões, com mecanismos que manipulam informações através de células recorrentes.

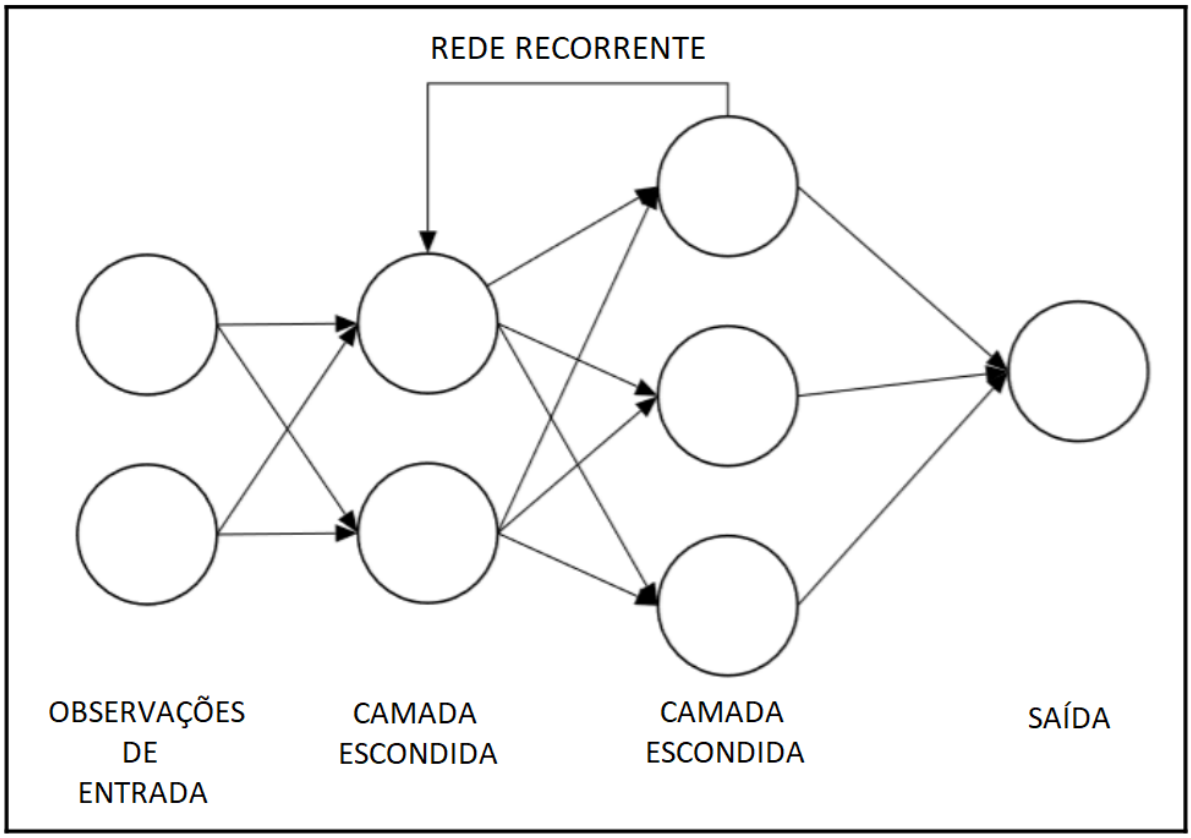

Figura 2.3: Exemplo de Rede Recorrente (Fig. traduzida de [23])

2.2.3 Redes de Memória de Longo-curto Prazo

Long Short-Term Memory (LSTM), em português Redes de Memória de Longo-curto Prazo é um tipo de RNN que foi projetada para evitar o problema da dependência de longo-termo [14].

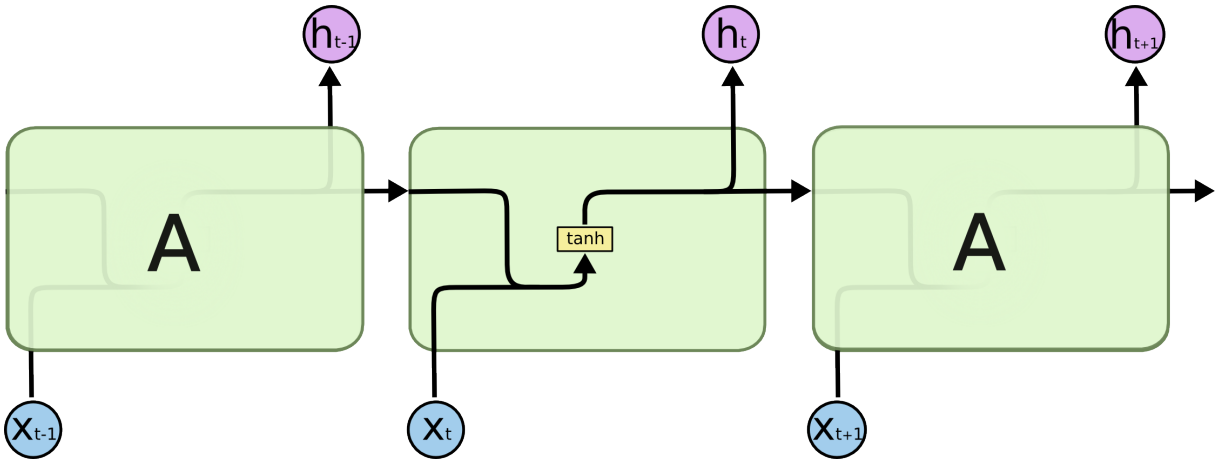

Figura 2.4: Módulo de repetição de uma RNN padrão (Fig de [34]).

A Figura 2.4 mostra a estrutura de uma RNN padrão A, que recebe um valor de entrada X_t e retorna como saída um valor h_t. Um *loop* permite que as informações sejam passadas de uma etapa da rede para a próxima. Essa RNN padrão, contém apenas uma camada com a função de ativação da tangente hiperbólica.

A estrutura de uma LSTM é um pouco diferente. Ao invés de possuir apenas uma camada, possui quatro camadas que interagem de forma singular. No diagrama da Figura 2.5, cada linha carrega um vetor inteiro, desde a saída de um nó até as entradas de outros. Os círculos rosa representam operações pontuais, como a adição de vetores, enquanto as caixas amarelas são camadas de redes neurais aprendidas. As linhas que se fundem denotam a concatenação, enquanto uma linha bifurcada denota que seu conteúdo está sendo copiado e as cópias vão para locais diferentes (Veja a legenda da Fig. 2.6).

A característica chave da LSTM é mostrada na linha superior horizontal do diagrama da Figura 2.5. Essa linha é o estado da célula e indica que as informações irão sofrer interações lineares e seguir adiante para a próxima iteração. As informações podem ou não sofrer pequenas alterações. A LSTM tem também a habilidade de adicionar ou remover informações do estado da célula através de portões (gates) que são representados por σ seguidos de $\times$ no diagrama. Portões são compostos de uma camada sigmoide e uma

operação de multiplicação. A camada sigmoide produz números entre 0 e 1, descrevendo quanto de cada componente deve ser deixado passar. Um valor de 0 significa "não deixe nada passar", enquanto um valor de 1 significa "deixe tudo passar".

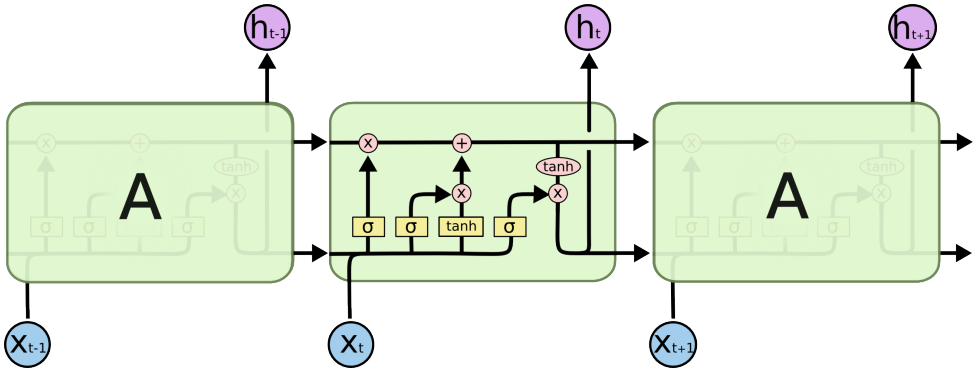

Figura 2.5: Módulo de repetição de uma LSTM que contém 4 camadas (Fig de [34]).

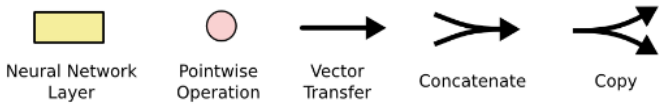

Figura 2.6: Legenda dos diagramas LSTM (Fig de [34]).

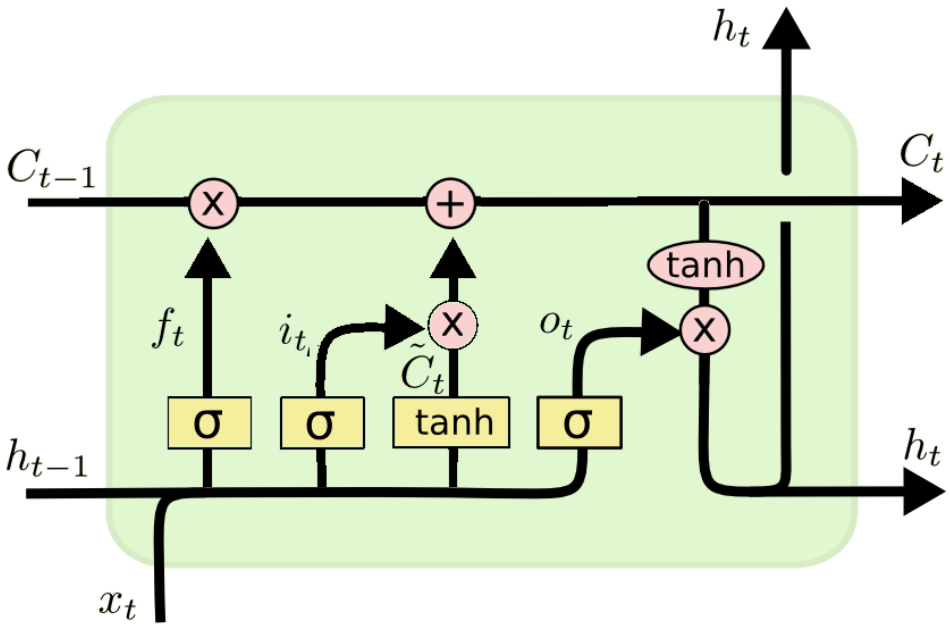

Figura 2.7: Célula Expandida de uma LSTM (Fig de [34]).

O fluxo, passo a passo, de uma LSTM é mostrado no diagrama da Figura 2.7 e, como explicado em [34], funciona da seguinte forma: O primeiro passo decide quais informações serão descartadas do início do estado da célula (C_t). Esta decisão é feita por uma camada sigmoide chamada de "forget gate layer", que gera um número entre 0 e 1 para cada número do estado da célula C_{t-1} a partir de h_{t-1} e x_t. Nos casos extremos, o valor 1 significa que é para manter a informação, enquanto que 0 significa que é para descartar a informação. A função f_t que faz a operação sobre estado C_{t-1} com h_{t-1} e x_t é mostrada na Equação 2.6.

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{2.6}$$

O próximo passo serve para descobrir quais novas informações serão armazenadas no estado da célula. Este passo é dividido em duas partes. A primeira, é uma camada sigmoide chamada "input gate layer" que decide quais valores serão atualizados (Equação 2.7). A segunda parte, é uma camada de tangente hiperbólica que cria um vetor de novos valores candidatos, $\tilde{C}_t$, que podem ser adicionados ao estado da célula (Equação 2.8). No próximo passo, essas duas partes são combinadas para criar uma atualização para o estado da célula.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{2.7}$$

$$\tilde{C}_t = tanh(W_c \cdot [h_{t-1}, x_t] + b_c) \tag{2.8}$$

Após obter os resultados anteriores, deve-se atualizar o estado da célula antiga C_{t-1} para o novo estado da célula C_t. Essa atualização é feita multiplicando o estado da célula antiga, $Ct - 1$, por f_t, que faz com que alguns resultados sejam esquecidos. Na sequencia são adicionados $i_t * \tilde{C}_t$, que são os novos valores candidatos, dimensionados arbitrariamente para atualizar cada valor de estado (Passo completo mostrado na Equação 2.9).

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \tag{2.9}$$

A última etapa é usada para decidir o que será produzido como saída. A nova saída será uma versão filtrada do novo estado da célula. Primeiro, uma camada sigmoide é aplicada para decidir quais partes do estado da célula serão usados como saída (Equação 2.10). A seguir, o estado da célula é colocado entre -1 e 1 através da função tanh e

o resultado é multiplicado pela saída da porta sigmoide, de modo que apenas saia como resultado as partes que foram decididas para passar (Equação 2.11).

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{2.10}$$

$$h_t = o_t * tanh(C_t) \tag{2.11}$$

LSTMs podem ser usadas em conjunto com outras redes neurais. O algoritmo PPO implementado no ML-Agents (Subseção 2.3) traz o recurso de poder ativar ou não as LSTMs em seu algoritmo.

2.2.4 PPO

Proximal Policy Optimization (PPO) [45], em português, Otimização de Política Proximal, é um algoritmo de RL que utiliza NNs, da família dos métodos de otimização de políticas, que utiliza várias épocas de gradiente ascendente estocástico para executar cada atualização de política. Além disso, o PPO tem a estabilidade e a confiabilidade dos métodos *trust-region* [44], mas é muito mais simples de ser implementado. O PPO é um algoritmo de RL livre de modelo (model-free), on-policy, seu espaço de ação e espaço de observação podem ser contínuos e utiliza a função de *Vantagem* (Ad) ao invés dos valores Q. Valores Q geralmente são utilizados no método de RL chamado Q-Learning, ele indica o valor de executar uma ação $a \in AA$ num estado $s \in S$. Já o operador de *Vantagem* é o valor Q ($Q(s, a)$) menos a média da função de valor no estado s ($V(s)$), ou seja, $Ad = Q(s, a) - V(s)$.

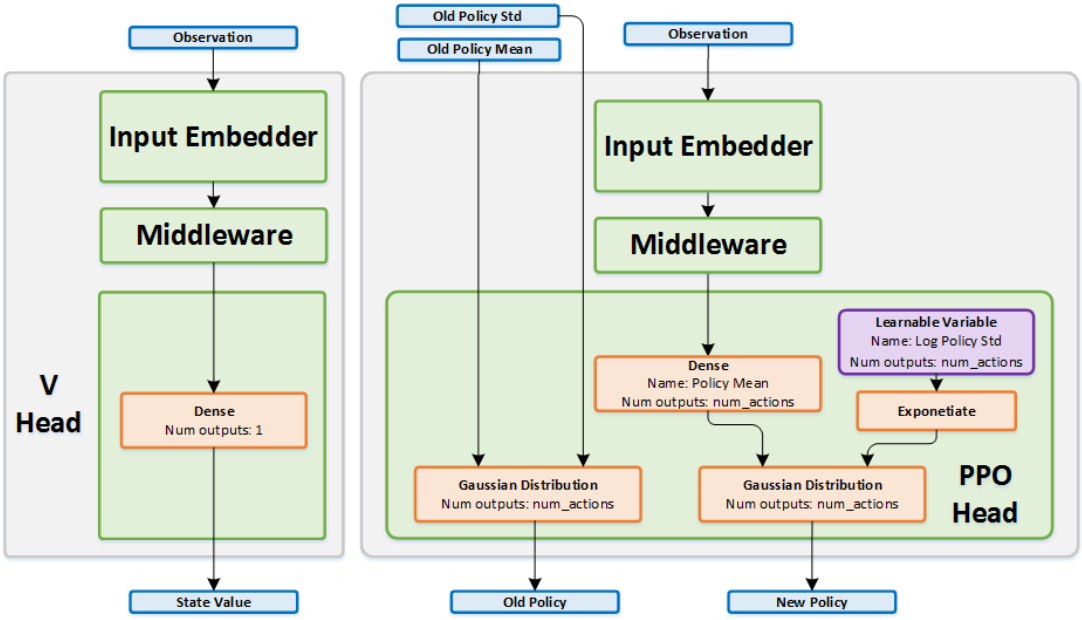

Figura 2.8: Estrutura da rede neural do PPO (Fig. do RL Coach[1]).

Na Figura 2.8 mostramos a estrutura de uma rede neural do PPO. Uma parte da rede calcula a função de valor e a outra parte a função de política. O *Input Embedder* é o primeiro estágio da rede, destinado a converter a entrada em uma representação de vetor de características. O *Middleware* obtém a saída do *Input Embedder* e o processa em um domínio de representação diferente, antes de enviá-lo pela *Output Head*, que é usado para prever os valores necessários da rede. Estes podem incluir valores de ação, valores de estado ou uma política. O PPO combina 2 *Output Heads*, um para política e outra para valor do estado. Mais informações sobre os componentes presentes nesta imagem podem ser encontrados na documentação do framework RL Coach.

Em métodos de gradiente de política, o estimador de gradiente de política mais utilizado é o da Equação 2.12, onde π_θ é uma política estocástica e $\hat{A}d_t$ é um estimador da função de vantagem no tempo t. Nessa equação, a expectativa $\hat{\mathbb{E}}_t[\dots]$ indica a média empírica sobre um lote finito de amostras, em um algoritmo que alterna entre amostragem e otimização [45].

$$\hat{g} = \hat{\mathbb{E}}_t[\nabla_\theta \; log\pi_\theta(a_t \mid s_t)\hat{A}d_t] \tag{2.12}$$

Ao invés de usar uma única rede neural para derivar os valores Q ou alguma forma de política, o algoritmo PPO usa uma técnica chamada *actor-critic*. Este método, em

[1]Documentação sobre as estruturas presentes no RL Coach: `https://nervanasystems.github.io/coach/design/network.html` e mais especificamente sobre as estruturas do PPO: `https://nervanasystems.github.io/coach/components/agents/policy_optimization/ppo.html`

essência, calcula tanto valores quanto políticas. No *actor-critic* são treinadas duas redes. Uma rede atua como uma estimativa ou crítico (critic) de valor Q, e a outra determina a política ou ações do ator (actor) ou agente [23]. O PPO não mais calcula os valores Q, e passa a gerar estimativas de recompensa (R) para que a Vantagem possa ser calculada (Equação 2.13). Destacamos também outra vantagem: um treinamento *actor-critic* foi desenvolvido para trabalhar com vários agentes assíncronos, onde cada agente tem seu próprio ambiente.

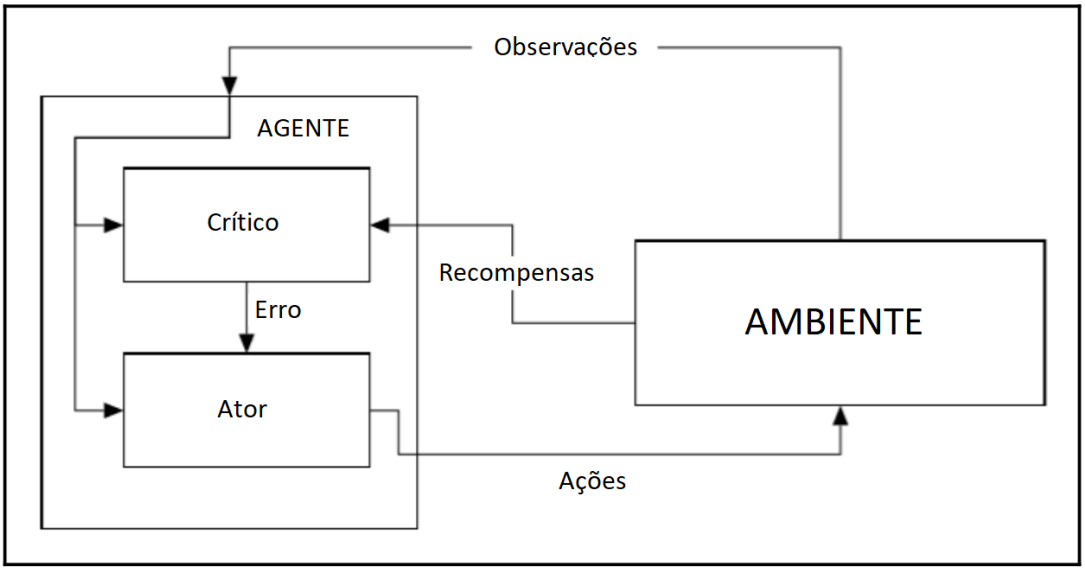

Figura 2.9: Diagrama de uma rede actor-critic (Fig. traduzida de [23])

$$Ad = R - V(s) \tag{2.13}$$

A Figura 2.9 mostra o diagrama de uma rede A3C e o termo Error que é comunicado entre o crítico (critic) e o ator (actor) é derivado das seguintes equações:

$$ValueLoss := \sum (R - V(s))^2 \tag{2.14}$$

$$PolicyLoss := -log(\pi(a|s)) * Ad(s) \tag{2.15}$$

Minimizamos o erro, calculando a entropia (Equação 2.16). A entropia mede a propagação de probabilidade. Enquanto uma entropia alta representa um agente com várias ações similares, o que dificulta as decisões do agente, uma entropia baixa nos diz que um

agente pode tomar decisões mais bem informadas [23].

$$H(\pi) = -\sum(P(x)log(P(x)) \tag{2.16}$$

Utilizando entropia, a função de perda de política (PolicyLoss) passa a ser a Equação 2.17, onde β corresponde à força da regularização da entropia (o que torna a política mais aleatória). Por fim, a função de perda final é a combinação entre as duas funções de perda já apresentadas (Equação 2.18). Esta é a função de perda que a rede neural tentará minimizar.

$$PolicyLoss := -log(\pi(a|s)) * Ad(s) - \beta * H(\pi) \tag{2.17}$$

$$L = 0.5 * \sum(R - V(s))^2 - log(\pi(a|s)) * Ad(s) - \beta * H(\pi) \tag{2.18}$$

A principal melhoria do PPO em relação ao método actor-critic [29] foi alterar a expressão usada para estimar o gradiente de política. Ao invés de utilizar a expressão do gradiente de probabilidade logarítmica da ação tomada, o método PPO usa a razão entre a política nova e a antiga dimensionada pelos valores de Vantagem [24]. O objetivo original do actor-critic pode ser escrito assim $J_\theta = \mathbb{E}_t[\nabla_\theta \ log\pi_\theta \ (a_t|s_t) \ Ad_t]$. Já o novo objetivo proposto pelo PPO é expresso como $J_\theta = \mathbb{E}_t[\frac{\pi_\theta \ (a_t|s_t)}{\pi_{\theta_{old}} \ (a_t|s_t)} \ Ad_t]$. Porém, se o PPO focasse em maximizar esse novo valor de maneira cega, uma atualização muito grande poderia acontecer nos pesos da política, e para solucionar esse problema, o objetivo é substituído pelo objetivo recortado (*clipped*). A razão entre a nova e a velha política podem ser representadas pela função:

$$r_t(\theta) = \frac{\pi_\theta \ (a_t|s_t)}{\pi_{\theta_{old}} \ (a_t|s_t)} \tag{2.19}$$

E o objetivo com recorte se torna a Equação 2.20 que limita a razão entre a nova e a velha política para um intervalo $[1 - \epsilon, 1 + \epsilon]$. Com essa mudança, é possível variar o ϵ para limitar o tamanho da atualização.

$$J_\theta = \mathbb{E}_t[\ min(r_t(\theta)Ad_t, \ clip(r_t(\theta), \ 1 - \epsilon, \ 1 + \epsilon)Ad_t) \ ] \tag{2.20}$$

Há ainda outra diferença entre A3C e o PPO com relação à estimativa do valor de Vantagem. No A3C, o valor de Vantagem é obtido a partir da estimativa do horizonte

finito dos passos T como $Ad_t = -V(s_t) + r_t + \gamma r_{t+1} + \cdots + \gamma^{T-t+1} r_{T-1} + \gamma^{T-t} V(s_T)$. Já o PPO usa uma estimativa mais generalista como a equação abaixo.

$$Ad_t = \sigma_t + (\gamma\lambda)\sigma_{t+1} + (\gamma\lambda)^2\sigma_{t+2} + \cdots + (\gamma\lambda)^{T-t+1}\sigma_{T-1} \tag{2.21}$$

$$onde\ \sigma_t = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{2.22}$$

A Vantagem do actor-critic passa ser apenas um caso especial da Vantagem do PPO onde o λ é igual a 1.

O método PPO também utiliza um procedimento de treinamento um pouco diferente, a Vantagem é estimada para toda uma sequência de amostras, antes de várias épocas de treinamento serem realizadas, quando uma longa sequência de amostras é obtida do ambiente [24]. Uma explicação mais detalhada pode ser encontrada no artigo original do PPO[45].

Com a RL, dependendo de um problema, um agente precisa de um treinamento longo para que ele consiga aprender a executar uma tarefa complexa. Uma forma de se obter um atalho para se aprender uma tarefa sem precisar conhecer as recompensas e sem precisar otimizar uma função de recompensa, é usar algoritmos de Aprendizagem por Imitação, em que um especialista demonstra para um agente aprendiz uma sequência de movimentos e esse consegue aprender a imitar esta sequência sem precisar saber as recompensas [16].

2.2.5 Aprendizagem por Imitação

A Aprendizagem por Imitação (IL) consiste em fazer com que um agente aprenda a executar uma tarefa a partir de *demonstrações de especialistas*, com amostras de trajetórias destes, sem sinal de reforço e sem dados adicionais do especialista enquanto treina [25]. Com (IL), tanto recompensas como políticas podem ser aprendidas a partir de demonstrações, almejando acelerar o tempo de treinamento para o aprendizado de alguma tarefa.

Existem duas abordagens principais para a IL: Clonagem Comportamental (BC) e Aprendizagem por Reforço Inversa (IRL). A IRL é o problema de extrair uma função de recompensa, dado o comportamento ótimo observado [32]. Apesar do nome, o *Apprenticeship Learning* abordado em [1], é uma abordagem de IRL. Behavioral Cloning (BC), em português, Clonagem Comportamental é um método de aprendizagem supervisionada que mapeia pares de estado/ação de trajetórias de especialistas para políticas, sem aprender a função de recompensa [25]. O BC também é conhecido como *Apprenticeship Learning* ou como Aprendizagem por Demonstração (*Learning from Demonstration*). Alguns artigos

criaram suas próprias versões do BC [40, 13, 55]. O BC funciona da mesma forma que a aprendizagem supervisionada para problemas de classificação de imagens, como também, para outras tarefas tradicionais de ML.

Nesta dissertação, utilizamos apenas o BC para realização de alguns treinamentos. A estrutura de sua rede neural pode ser observada na Figura 2.10. O BC funciona da seguinte forma, o agente:

- observa o estado atual e a ação escolhida pelo especialista.
- escolhe uma ação baseada numa política que será aprendida (inicialmente esta política é aleatória).
- compara sua ação escolhida com a demonstrada pelo especialista por meio de uma função de erro
- tenta otimizar sua política utilizando a função de erro
- uma nova iteração é iniciada e todo o processo é repetido.

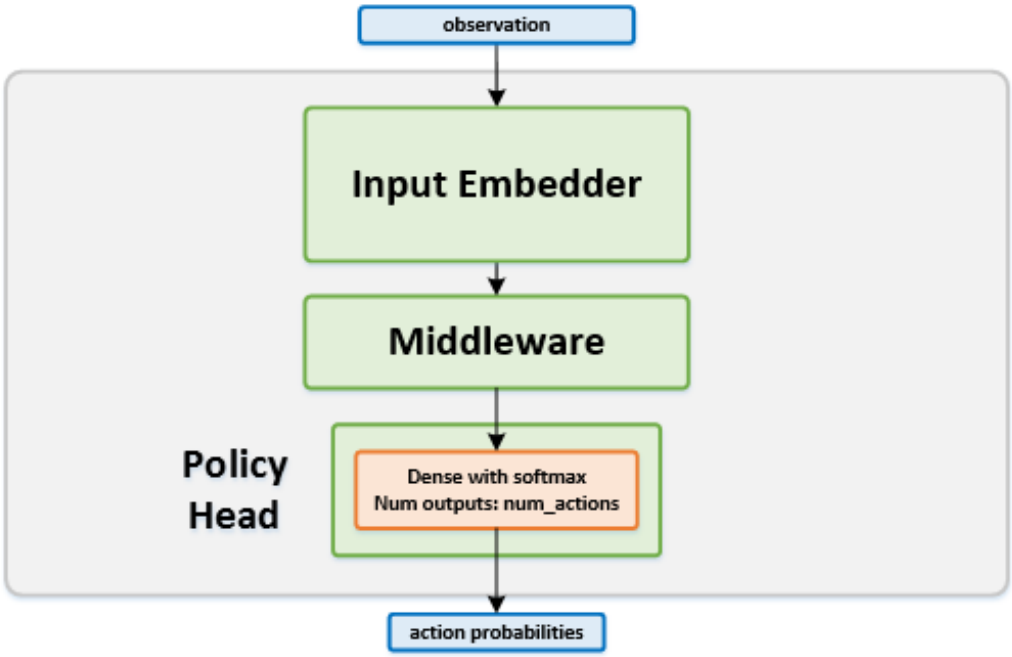

Figura 2.10: Estrutura da rede neural do BC. Os componentes básicos desta imagem foram explicados na Figura 2.8 da estrutura da rede neural do PPO. Mais informações podem ser encontradas no RL Coach[3].

Pesquisas abrangentes sobre diferentes métodos de Aprendizagem por Imitação podem ser encontradas nos artigos [2, 16].

2.3 Kit de ferramentas ML-Agents

O Unity Machine Learning Agents Toolkit é um projeto de código aberto que permite que pesquisadores e desenvolvedores criem ambientes de simulação usando o Editor da Unity e interajam com eles a partir de uma API Python [18]. O pacote Python traz a implementação dos algoritmos de ML. Este pacote interage com o ambiente do jogo que pode ser o executável ou o próprio editor da Unity.

Unity, também conhecida como Unity3D, é um motor de jogo e um ambiente de desenvolvimento integrado (IDE) para a criação de mídia interativa, tipicamente jogos digitais [11]. Os autores do artigo [18] fizeram um bom resumo da terminologia básica da Unity. Um Projeto da Unity é composto por uma coleção de *Assets*, que tipicamente correspondem a arquivos dentro do Projeto. *Scenes* são um tipo especial de *Asset* que definem um ambiente ou níveis de um Projeto/Jogo. *Scenes* contém uma definição da composição hierárquica dos *GameObjects*. Estes, correspondem aos objetos reais, físicos ou puramente lógicos, dentro do ambiente. O comportamento e função de cada *GameObject* é determinado pelos componentes anexados a ele. Há uma variedade de componentes internos fornecidos com o Editor da Unity, incluindo Câmeras, Malhas, *Renderers*, *Rigid-Bodies* e muitos outros. Também é possível definir componentes personalizados usando *scripts* C# ou *plug-ins* externos.

Após a importação do SDK do ML-Agents para dentro de um Scene do editor da Unity, um ambiente de aprendizado é criado. Nesse ambiente há três tipos de entidades: Agents (agentes), Brains (cérebros) e Academy (academia). Os Agents são responsáveis por coletar observações e realizar ações. Cada Brain é responsável por fornecer uma política de tomada de decisão para cada um dos agentes associados. E a Academy é responsável pela coordenação global da simulação do ambiente. Na Figura 2.11 é possível observar a hierarquia do ML-Agents. O componente Academy é pai de todos os Brains e responsável por se comunicar com a API Python que não necessariamente precisa utilizar TensorFlow. Um Brain pode fornecer política para um ou mais Agents. Além disso, há quatro tipos de Brains: heurísticos, internos, externos, e controlados pelo jogador. Para treinar um Brain com o pacote Python, o tipo deve ser externo. Este tipo também pode ser utilizado por

[3]Veja a estrutura da rede neural do BC criada no framework RL Coach `https://nervanasystems.github.io/coach/components/agents/imitation/bc.html`

qualquer algoritmo externo feito em Python (não é obrigado utilizar TensorFlow). Para utilizar um modelo já treinado com TensorFlow, o tipo deve ser interno. O tipo heurístico é usado quando se programa em linguagem C# o comportamento que determinado agente deve seguir. Quanto ao tipo controlado pelo jogador, o agente é controlado via dispositivos de entrada pelo próprio jogador.

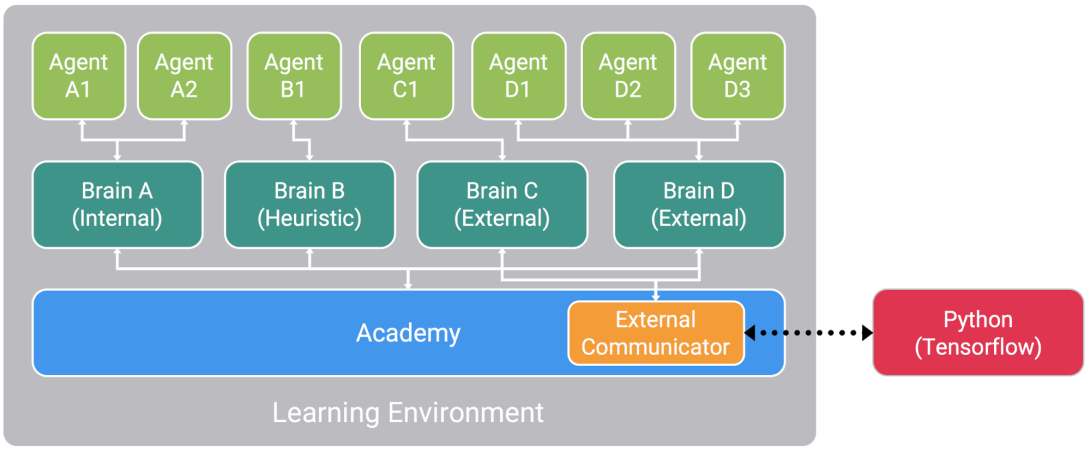

Figura 2.11: Hierarquia de um Ambiente de Aprendizagem do ML-Agents. (Fig de [53])

Para se criar um novo agente com o ML-Agents, uma classe que herde de Agent precisa ser criada e o método CollectObservations() deve ser implementado para informar as observações, e o método AgentAction() para informar as recompensas de acordo com as ações de um agente.

As observações de um Brain podem ser visuais e/ou por meio de um vetor de observações. As observações são visuais quando imagens capturadas das câmeras ou qualquer outro tipo de imagem é utilizada como observação. As ações podem ser discretas ou contínuas. Como falamos anteriormente, um ambiente de aprendizado pode ter um ou mais Brains. Quando há apenas um Brain externo no ambiente, dizemos que é um treinamento de apenas um Brain (ou treinamento Um-Brain), se há mais de um Brain dizemos que é um treinamento de múltiplos Brains (ou treinamento Multi-Brain). No Capítulo 5, na Seção 5.3, investigamos qual é a melhor forma de se treinar um Brain, se através de um treinamento One-Brain ou Multi-Brain.

Uma tradução livre resumida do artigo [18], realizada por nós, que contém uma explicação mais detalhada do funcionamento do ML-Agents pode ser encontrada no Anexo A.

O repositório do ML-Agents Toolkit é fornecido via o sistema aberto de versionamento GitHub[4] assim como a sua documentação[5]. Recomendamos a leitura do artigo [18] e

[4]https://github.com/Unity-Technologies/ml-agents
[5]https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Readme.md

do livro [23] para se obter informações mais detalhadas da arquitetura do ML-Agents Toolkit, mais exemplos práticos e do por quê da Unity ser uma plataforma robusta para os pesquisadores e desenvolvedores de algoritmos de ML.

A versão do ML-Agents utilizada nesta pesquisa foi a versão 0.5. Nesta dissertação, algumas pequenas alterações foram feitas no SDK que é importado para dentro da Unity e também na API Python. No SDK, alteramos o código relativo ao Brain Heurístico para que fosse possível recuperar a ação lida de um arquivo de repetição durante o treinamento do agente que utilizou o algoritmo BC (CoreBrainHeuristic.cs foi alterado para informarmos o número do agente e o cenário no qual ele está jogando). Alteramos o código relativo às condições de reinicialização da classe Agent caso o agente entre no estado de Finalizado (Done). Antes o agente era reiniciado automaticamente quando entrava no estado Finalizado, mesmo caso nossa classe controladora (MapController) não requisitasse uma nova atualização e isso gerava alguns erros de sincronia. Ainda na classe Agent, criamos uma nova função que força o estado do Agente a ir para Finalizado, com isso podemos aplicar uma recompensa negativa de morte ao agente e finaliza-lo ao mesmo tempo caso ele esteja dentro de sua função que informa as observações para o Brain, o que é útil pois verificamos se um agente foi atingido por fogo tanto na hora da observação quanto após efetuar uma ação. Se o agente é atingido na hora da observação, então ele deve ser finalizado antes de requisitar uma nova ação.

Na API Python, fizemos 2 alterações no arquivo trainer_controller.py. 1) Acrescentamos uma nova forma de continuar um treinamento com um modelo já treinado. Esta forma força o carregamento de um modelo anterior fazendo uma interseção de todos os tensores do TensorFlow que contém o mesmo nome. Caso haja o mesmo tensor tanto no modelo antigo quanto no novo, os valores deste tensor são recuperados do modelo antigo, senão os valores são inicializados da forma padrão (a variável newLoadMode ativa ou desativa este modo). 2) Criamos uma variável booleana continueTrainWithPPOAfterBC que informa se o modelo previamente treinado pelo BC, que deve estar dentro da pasta "./models/bc_train", será usado ou não para iniciar um novo treinamento. Caso não seja utilizado, e o treinamento foi configurado para carregar o modelo antigo para continuar o treinamento, o ML-Agents buscará o modelo antigo na mesma pasta onde será gerado o novo modelo.

2.4 Bomberman Original

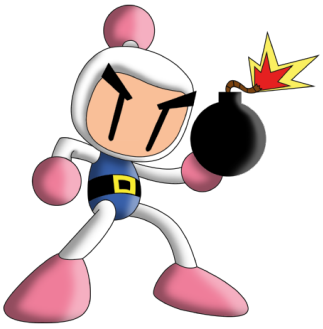

Figura 2.12: Arte do personagem principal do jogo Bomberman.

O jogo Bomberman [50], em alguns lugares, conhecido também como DynaBlaster, é uma franquia de jogo de estratégia baseado em labirintos que foi originalmente desenvolvido pela empresa Hudson Soft em 1985. Desde essa data, 99 versões de Bomberman já foram lançadas de acordo com o sítio web uvlist.net[6]. Algumas capturas de tela do primeiro jogo são mostradas nas Figuras 2.13 e 2.14. Diversos vídeos demonstrando a jogabilidade do jogo estão disponíveis online[7]. Além disso, a comunidade de jogadores e seguidores de Bomberman continua grande, chegando a quase 1 milhão de seguidores no facebook[8]. Ademais, seus seguidores numa plataforma de wiki já publicaram mais de 1583 páginas no Bomberman Wiki[9].

Figura 2.13: Tela de menu do Bomberman de 1985

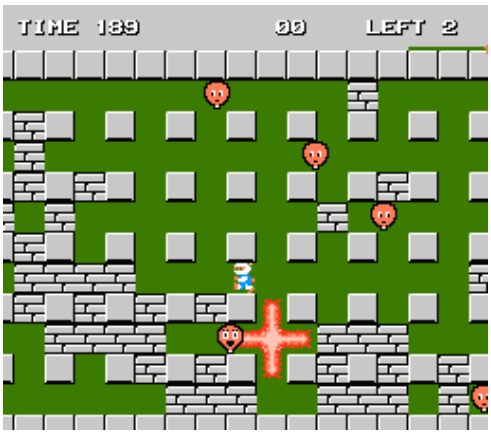

Figura 2.14: Captura de tela do jogo demonstrando a jogabilidade.

[6]https://www.uvlist.net/groups/info/bomberman
[7]https://www.youtube.com/watch?v=o1uTr_7xqPI
[8]https://www.facebook.com/pg/officialbomberman/community/
[9]http://bomberman.wikia.com/wiki/Main_Page

O objetivo do jogo Bomberman é colocar bombas estrategicamente pelo cenário para matar inimigos ou destruir blocos do cenário com o objetivo de abrir caminhos, ou de encontrar itens de poderes. Inclusive, a maior parte dos jogos dessa franquia traz o modo multijogador, onde o objetivo é ser o último jogador vivo no cenário, matando todos os oponentes presentes, que podem ser controlados por outros jogadores ou pelo próprio jogo. Independentemente do modo de jogo, a jogabilidade básica do Bomberman continua a mesma. Todo cenário do jogo é um retângulo cercado por blocos indestrutíveis, e, geralmente, as primeiras linhas e colunas que são vizinhas desses blocos indestrutíveis ficam vazias ou possuem blocos destrutíveis. Após essas duas camadas que circundam o cenário, há blocos indestrutíveis, com células intercaladas formando o cenário básico de cada fase do jogo Bomberman. Todas as células que não possuem blocos indestrutíveis podem possuir blocos destrutíveis que ficarão atrapalhando o caminho que provavelmente deverá ser aberto. Além disso, estes blocos, ao serem destruídos por bombas, podem gerar poderes que trazem alguma melhoria para o agente (e.g. colocar mais bombas, aumentar o alcance do fogo das bombas e etc). As bombas colocadas pelos jogadores possuem um certo alcance e quando explodem numa célula se propagam vertical e horizontalmente para as células vizinhas, respeitando esse alcance e obstruções no caminho.

Algumas das características que tornam o Bomberman uma excelente plataforma para se desenvolver e testar novos algoritmos e métodos para a construção de agentes inteligentes são: o gráfico 2D simples, o ambiente dinâmico, o raciocínio estratégico, o modo multijogador, os poderes, os novos mapas, busca por caminhos, posicionamento estratégico de bombas, e sobrevivência em ambientes hostis [5]. Além dessas, há outras características interessantes, como: jogabilidade simples e intuitiva, planejamento de estratégias em equipe ou individuais, reconhecimento da importância dele na comunidade de jogos, e a recompensa atrasada.

Capítulo 3

Trabalhos Relacionados

Neste capítulo são apresentados os trabalhos relacionados que inspiraram e influenciaram o desenvolvimento desta pesquisa de dissertação.

3.1 BAIP - Bomberman como uma Plataforma de Inteligência Artificial

Bomberman como uma Plataforma de Inteligência Artificial (BAIP) [5] é uma plataforma gráfica, de código aberto, agnóstica à linguagem, que tem como meta ajudar no estudo e desenvolvimento de novas técnicas de AI, sendo baseada no jogo Bomberman [50]. A plataforma inclui métodos heurísticos de busca, planejamento e de RL para criar agentes inteligentes para jogar Bomberman. Contudo, os agentes que utilizam técnicas de RL não foram capazes de jogar a mesma versão do jogo que os agentes heurísticos e de busca. Os agentes de RL foram testados num exemplo de *Grid World*, que é um cenário simples formado por uma grade retangular 2D de tamanho arbitrário com um agente que inicia em um quadrado da grade e tenta mover para outro quadrado da grade localizado em outro lugar que é seu objetivo.

O autor desenvolveu algumas heurísticas para o agente conseguir andar pelo cenário, destruindo blocos, evitando explosões e matando seus adversários. Por conta disso, seu agente simples, baseado nessa heurística, se demonstrou apto a jogar o jogo.

O BAIP inclui, ainda, agentes baseados em busca e planejamento capazes de gerar uma ação numa árvore de decisão e explorar eficientemente o espaço de estados. Agentes baseados na busca em largura (BFS), em profundidade (DFS) e no A* (A estrela) foram criados. Esses agentes demonstraram ser superiores aos agentes heurísticos.

O autor deixou em aberto o campo dos algoritmos de RL que utilizam redes neurais para resolver o problema do Bomberman. Esta foi uma de nossas motivações para que o estudo realizado a respeito de agentes inteligentes baseados em RL via NNs para jogar Bomberman fosse produzido. Optamos por desenvolver nosso próprio ambiente de aprendizagem do Bomberman em um motor de jogo mais famoso e que possuísse uma comunidade de desenvolvedores mais ativa, que é o caso do motor de jogo Unity3d[54].

3.2 Métodos de Exploração para o Q-Learning Conexionista em Bomberman

Em [21] os autores utilizaram um agente representado por uma MLP que aprende a jogar o jogo com o uso do Q-Learning. Eles apresentam novos métodos de exploração que obtiveram um melhor desempenho no jogo Bomberman, entre os métodos de exploração investigados. Foram produzidas duas novas estratégias de exploração: Error-Driven-ϵ e Interval-Q, que baseiam seu comportamento exploratório no erro de diferença temporal do Q-learning. Estas duas técnicas foram comparadas com outras cinco técnicas: Random-Walk, Greedy, e-Greedy, Diminishing e-Greedy, e Max-Boltzmann.

O agente desenvolvido utilizou perceptron multicamadas (MLP) para aprender o jogo através do Q-Learning, o que se assemelhou bastante ao agente desenvolvido por nós. Nesta dissertação utilizamos o algoritmo PPO que também utiliza uma MLP quando imagens não são utilizadas como observação. Entretanto, MLP com Q-Learning é uma técnica simples comparada ao PPO, que é um algoritmo de gradiente de política do estado da arte que utiliza o framework actor-critic e que pode fazer treinamentos assíncronos com vários agentes treinando de uma vez só [24]. Infelizmente, os autores dessa pesquisa não disponibilizaram o código fonte.

O trabalho traz uma boa contribuição para nossa pesquisa, porque trouxe um exemplo de como modelar a representação do estado do jogo Bomberman num vetor (de características) de entrada para alimentar a MLP. Ele também apresenta algumas regras básicas, tais como: agentes podem ocupar a mesma célula no cenário e o que fazer quando o limite de iterações é alcançado. Nesta última situação, o jogo começa a gerar bombas pelo cenário até que pelo menos um agente sobreviva. Além disso, o artigo menciona que todos os agentes devem observar o estado atual do jogo simultaneamente e executar suas ações após as observações também simultaneamente, para que nenhum agente tire vantagem sobre o outro. O kit de ferramentas ML-Agents, que foi utilizado nesta dissertação, tam-

bém executa as ações dos agentes simultaneamente, sendo que todos os agentes observam o estado atual primeiro para depois executarem suas ações (e.g. $o1 \rightarrow o2 \rightarrow a1 \rightarrow a2$, i.e. agente 1 observa o ambiente, agente 2 observa o ambiente, agente 1 age no ambiente, agente2 age no ambiente).

3.3 Uma Implementação de um Agente Autônomo para o Jogo Bomberman com RL

Em [28], os autores desenvolveram um agente inteligente para jogar Bomberman através do algoritmo Q-Learning, tendo sido bem sucedido apenas numa versão simplificada do jogo onde o adversário ficava imóvel. Os autores chegaram a conclusão que o espaço de estados da versão que possui adversários dinâmicos do jogo Bomberman é muito vasto, o que não permite que o algoritmo Q-Learning consiga convergir ou armazenar tanta informação.

A tentativa de criar um agente inteligente com Q-Learning puro reforça a premissa de que para se conseguir desenvolver uma inteligência artificial baseada apenas em RL, sem busca e planejamento, é necessário utilizar alguma aproximação por funções, como as redes neurais artificiais, pelo fato do espaço de estados ser muito vasto.

3.4 Métodos Reflexivo, MiniMax e Q-Learning para Bomberman

Em [7], os autores fazem um estudo comparativo entre 3 categorias de agentes: Reflexivos, MiniMax e Q-Learning para um jogo estilo Bomberman em turno e discreto. Eles conseguiram desenvolver um algoritmo MiniMax que derrota um jogador humano de forma consistente, porém, é computacionalmente caro e incapaz de rodar em tempo real. Por isso, desenvolveram o agente Reflexivo que também pode derrotar jogadores humanos, na maioria das vezes, mas não consegue vencer o agente MiniMax. Por fim, a tentativa deles de criar um agente Q-Learning não foi bem sucedida, sendo o melhor resultado obtido para um cenário estático sem blocos destrutíveis e com apenas um inimigo parado. O agente Q-Learning aprendeu a colocar bombas próximo ao inimigo para matá-lo. Já num cenário com blocos destrutíveis, o agente não conseguiu aprender que bombas são capazes de destruir esses blocos, pois quando o agente tenta destruir blocos, e morre mais vezes por conta disso do que consegue de fato destruí-los, ele aprende que a ação de colocar

bombas não é uma ação promissora. Os autores pensam que a recompensa atrasada da ação de colocar bomba e seus efeitos após algumas iterações a frente é o principal fator que contribuiu para que o agente não aprendesse a usar as bombas. Então, num cenário com blocos destrutíveis, o agente Q-Learning sempre aprende que não se deve usar bombas porque coloca sua própria vida em risco.

Um problema semelhante ocorreu na nossa pesquisa. Inicialmente o jogo Bomberman produzido era estocástico, tanto no tempo quanto no espaço. Nesta situação, a recompensa atrasada era um dos principais problemas a serem tratados. Talvez o tempo (de 25 turnos) utilizado por eles para que a bomba explodisse, foi um tempo muito longo. No nosso Bomberman, por exemplo, a bomba leva apenas 6 iterações para explodir, o que acaba facilitando a propagação da recompensa para os estados posteriores.

3.5 Pommerman

Pommerman [41] é um ambiente multiagente baseado no jogo Bomberman [50]. Esse ambiente consiste em um conjunto de cenários, cada um com pelo menos quatro jogadores e contendo aspectos cooperativos e competitivos, implementado na linguagem Python. Além disso, ele pode ser utilizado para se criar um conjunto diversificado de ferramentas e métodos, incluindo planejamento, modelagem de oponente/companheiro de equipe, teoria dos jogos e comunicação, e, consequentemente, pode servir como um benchmark de múltiplos agentes. Atualmente há 3 tipos de cenários: cada um por si, dupla versus dupla sem comunicação, dupla versus dupla com comunicação por rádio.

Pommerman foi utilizado já em duas competições, sendo a última na pista de competição NIPS 2018[1]. Competições envolvendo Pommerman são interessantes porque alavancam o estado da arte quanto ao desenvolvimento de novos agentes inteligentes.

Os cenários do Pommerman sempre possuem 11 linhas por 11 colunas. Agentes não podem ocupar a mesma célula, diferente do jogo original (Bomberman). As bombas demoram 10 iterações para explodir. Blocos destrutíveis podem gerar itens de poderes ao serem destruídos. Há 3 poderes: bomba extra (aumenta a munição), aumento de força da bomba (aumenta alcance do fogo da bomba), e poder de chute de bomba. Esse poder de chute permite que um agente chute bombas em uma direção. Isso é feito andando em direção a elas. A bomba então viaja na direção em que o agente estava se movendo a uma velocidade de uma unidade por iteração de tempo até que eles sejam impedidos por um

[1] https://nips.cc/Conferences/2018/CompetitionTrack

jogador, uma bomba ou uma parede.

As ações que o agente pode fazer no Pommerman são: 1) Ficar parado; 2) Mover pra cima; 3) Mover pra esquerda; 4) Mover pra baixo; 5) Mover pra direita; 6) Colocar bomba. Já as observações recebidas pelo agente são:

1. Tabuleiro 11×11: 121 inteiros. Cada célula é um inteiro apenas. Não há combinação de estados na mesma célula (e.g. O estado Agente + Bomba não é representado). Para os cenários de time, todas as células que não são visíveis pelo agente são preenchidas com valor de Neblina (5).
2. Posição: posição do agente na grade (x, y). Dois inteiros no intervalo [0,10]
3. Munição: quantidade máxima de bombas. Valor inicial é 1.
4. Força da explosão: inicialmente tem alcance 1 de distância.
5. Chute: agente pode chutar ou não a bomba (0 ou 1). Chute é dado quando agente anda em direção a bomba. É um poder que pode ser obtido no cenário.
6. Companheiro de equipe: 1 inteiro no intervalo [-1, 3]. Informa qual agente é o companheiro de equipe. Se o cenário for cada um por si, valor é -1.
7. Inimigos: 3 inteiros no intervalo [-1, 3], informando quais agentes são os inimigos. Se for um cenário de time, o ultimo inteiro é -1 para refletir que há somente 2 inimigos.
8. Força das Bombas: lista de 121 inteiros que informa a força da bomba presente em cada célula da grade. A regra da neblina se aplica aqui.
9. Tempo das Bombas: lista de 121 inteiros que informa o tempo que falta para cada bomba explodir em cada célula da grade. A regra da neblina também se aplica aqui.
10. Mensagem: Utilizado apenas para cenário de time com rádio. Contem uma lista de dois inteiros, sendo cada num no intervalo [0, 8]. A mensagem é retransmitida para companheiros de equipe. Ambos os inteiros são zero quando um companheiro está morto ou é o primeiro passo. Caso contrário, eles estão em [1, 8].

As recompensas no Pommerman são dadas apenas no final do episódio, ou seja, as recompensas são esparsas. Isso pode dificultar a aprendizagem de agentes de RL. Porém, recompensas artificiais podem ser criadas por quem está implementando um novo agente de RL. No nosso Bomberman, criamos recompensas em todas as iterações de uma partida.

A documentação detalhada das obervações recebidas pelo agente podem ser encontradas nesse link[2]. Além disso, o sítio web principal do Pommerman encontra-se no link[3].

Vale ressaltar que o ambiente desenvolvido nesta dissertação (BLE) trouxe algumas vantagens tais como: 1) o BLE foi desenvolvido utilizando-se o motor de jogo Unity [54], facilitando novas extensões e incrementos. 2) células com mais de um estado conseguem ser representadas e agentes podem ocupar a mesma célula, como no jogo original. 3) ainda que no BLE não haja poderes e times, essas funcionalidades podem ser desenvolvidas sem grandes esforços.

3.5.1 Agentes criados para o jogar o Pommerman

Novos métodos criados pela comunidade científica foram testados dentro do ambiente Pommerman. Em [37], os autores propuseram uma abordagem de busca em árvore em tempo real, onde esta busca é realizada apenas com uma profundidade limitada, e as folhas são avaliadas sob um cenário determinístico e pessimista. Eles utilizaram o modo de batalha de times do Pommerman para testar sua nova abordagem.

Em [56], os autores investigaram alguns algoritmos de busca em árvore para controlar as ações dentro dos estados de ataque e evasão da máquina de estados finitos do agente. Nos outros estados, o agente usava heurísticas para escolher as ações. Foi utilizado o modo cada um por si do Pommerman.

Em [46], os autores apresentaram e avaliaram algoritmos e estratégias para agentes independentes e coordenados (modo de times) dentro do Pommerman. No modo cada um por si, eles criaram um agente PPO, porém ele não conseguiu obter um desempenho significativamente melhor do que um agente baseado em regras. O agente não aprende a usar bombas, ele apenas aprende a não morrer ao invés de tentar vencer uma partida. Neste mesmo modo, eles criaram um agente DQfD (Deep Q-Learning from Demonstrations [12]) baseado em Aprendizagem por Imitação e usaram um agente baseado em regras para treinar o agente DQfD por demonstrações. Este agente ficou significativamente melhor que o agente PPO e pode explorar o mapa, colocar bombas e criar estratégias. No modo com times, eles discutiram estratégias cooperativas e não cooperativas.

Em [19], os autores apresentaram uma estrutura que usa um demonstrador simulado não especializado dentro de um método RL assíncrono distribuído para ter sucesso em domínios de exploração difícil. Vários experimentos foram feitos variando a qualidade e

[2]https://github.com/MultiAgentLearning/playground/blob/master/docs/environment.md
[3]https://www.pommerman.com/

o número de demonstrações. Eles simplificaram o jogo, diminuindo o tamanho da grade para 8 linhas por 8 colunas e permitiram apenas 2 agentes por partida.

Em [27], os autores desenvolveram um novo método, o MAGNet, para a RL multi-agente via NNs, incorporando informações sobre a relevância de outros agentes e objetos do ambiente para o agente MAGNet. O agente MAGNet superou todos os competidores (DQN [30], MCTSNets [10], MADDPG [26] e um agente heurístico que já vem implementado no Pommerman) no modo de time.

Geralmente os agentes que são mais vitoriosos no Pommerman, são os agentes que utilizam técnicas combinadas com algoritmos de árvores de decisão. Em nossa pesquisa, implementamos apenas agentes que utilizam algoritmos de RL e IL.

3.6 Beating Bomberman with Artificial Intelligence

Em [31] os autores propõem uma combinação de uma árvore de comportamento e um algoritmo de busca por caminho em um processo de tomada de decisão. Eles tem o objetivo de resolver o problema baseado em labirintos utlizando o jogo digital Bomberman da plataforma Nintendo Entertainment System (NES) por meio de um simulador chamado FCEUX. O emulador permite a codificação de algoritmos de AI dentro do jogo, permitindo que o computador jogue autonomamente. Para a busca por caminho, os autores usaram os algoritmos de busca em largura (BFS) e o A* (A estrela) para verificar a viabilidade dessas técnicas de IA nos jogos digitais. Este método se demonstrou robusto e o agente foi capaz de completar as fases do jogo. Essa versão do Bomberman não apresenta modo multijogador (batalha).

Como mencionado anteriormente, nossa pesquisa não usa informação explicitamente privilegiada do jogo Bomberman. Os dados utilizados para o aprendizado dos nossos agentes são desprovidos de informações adicionais. Queremos testar se nosso agente consegue aprender a jogar o Bomberman recebendo apenas a representação do estado da grade do tabuleiro do jogo e a posição X,Y do agente corrente.

Capítulo 4

Ambiente de Aprendizado Bomberman com RL

Neste capítulo apresentamos o Ambiente de Aprendizado Bomberman (BLE - Bomberman Learning Environment), construído com base nos algoritmos do ML-Agents toolkit: DRL e IL. O jogo Bomberman incluído como parte do BLE se assemelha ao Bomberman [50] original, mas aqui escolhemos focar no comportamento do agente Bomberman em relação aos outros agentes, ao cenário e ao seu desejo de permanecer vivo, deixando de lado habilidades extras de poder e não permitindo mais de uma bomba por agente. Com isso, o agente tem como meta ser o último agente vivo em cena a vencer a partida. O BLE foi desenvolvido dentro do motor de jogo Unity3D (versão 2018.2.7f1 Personal) e com ML-Agents Toolkit (versão 0.5). Seu repositório encontra-se no GitHub[1]. Nosso repositório contendo o ML-Agents, com algumas modificações feitas pós nós, também pode ser encontrado no GitHub [2]. Os principais Assets utilizados no BLE foram baixados do tutorial "How To Make A Game Like Bomberman With Unity" [8].

Na Seção 4.1 explicamos como o jogo nessa dissertação funciona. Posteriormente, formalizamos tarefa de aprendizado como um MDP na Seção 4.2. Na Seção 4.3 explicamos as diferentes maneiras de representar o estado do ambiente do jogo para os agentes. Em seguida, os algoritmos usados para treinar nossos agentes são informados na Seção 4.4. Finalmente, o *loop* do processo de treinamento é descrito na Seção 4.5

[1]Repositório público disponível no GitHub `https://github.com/lorel-uff/pip`
[2]Nosso *fork* da versão 0.5 do ML-Agents `https://github.com/icaro56/ml-agents`

4.1 Dinâmica do Jogo

O jogo do Bomberman criado como parte do BLE segue uma representação discreta e finita de tempo e espaço, onde o tempo é medido de acordo com o número de interações entre o agente e o ambiente de jogo e o cenário é dividido em um número discreto de células dentro de um tabuleiro em grade 2D. Cada cenário do jogo tem 9 linhas por 9 colunas e pode conter entre dois a quatro jogadores batalhando entre si. O BLE inclui cenários que podem ser compostos por 6 tipos de entidades: (i.) blocos indestrutíveis, (ii.) blocos destrutíveis, (iii.) agentes, (iv.) bombas, (v.) explosões das bombas, e (vi.) zonas de perigo. As zonas de perigo são entidades invisíveis que informam a extensão das explosões das bombas quando elas explodem.

No início de uma partida, os agentes são posicionados aleatoriamente em uma das quatro quinas possíveis, cada um em uma posição diferente. Em cada iteração do jogo, o agente recebe a observação do ambiente e realiza uma ação. O agente não recebe nenhuma observação privilegiada que facilite sua tomada de decisão, como o caminho para o inimigo mais próximo, ou o caminho para o bloco mais próximo, ou o caminho para chegar a uma posição segura, *etc.*. Informações privilegiadas não são dadas ao agente porque jogadores humanos também não as recebem.

Os agentes podem colocar bombas no cenário em sua posição atual. Depois de N iterações que a bomba foi colocada, ela explode, com um alcance de duas células para cada direção (para cima, para baixo, esquerda e direita). O agente só pode colocar uma bomba de cada vez e ele deve esperar pela explosão desta bomba antes de colocar outra. O fogo de uma explosão de bomba é configurado para durar apenas uma iteração.

Algumas entidades dentro do alcance de uma explosão obstruem seu efeito além delas. Assim, a explosão da bomba não passa pelos blocos indestrutíveis, outras bombas ou blocos destrutíveis. Por causa disso, na presença de tais entidades, a explosão de uma bomba é limitada a uma célula na direção do bloco (ou outra bomba) se estiver do mesmo lado da bomba. No entanto, observe que o fogo de uma bomba causa a explosão de outras bombas dentro do alcance de duas células.

Uma partida termina quando há apenas um agente vivo no cenário, ou quando todos estão mortos (empate). O último agente vivo no cenário é o vencedor do jogo. Para evitar que o jogo fique executando indefinidamente, também forçamos o jogo a terminar após 300 iterações, por adicionar bombas distribuídas aleatoriamente no cenário. Tais bombas são criadas após $N + 1$ iterações terem passado desde a última explosão. O processo de

geração de bombas começa com uma bomba e continua aumentando após cada explosão, limitado pelo número máximo de células livres que podem receber uma bomba. Optamos por gerar bombas após 300 iterações em vez de terminar o episódio e sortear um vencedor, para permitir que os agentes aprendam a fugir de bombas em situações bem extremas e para deixar a partida mais próxima da realidade do modo batalha do jogo original, em que o cenário vai se fechando, bloco a bloco, a partir das bordas, circundando no sentido horário em direção ao o centro do tabuleiro, gerando blocos indestrutíveis nestas posições até completar todo cenário com blocos ou sobrar apenas um jogador. Ao gerar um bloco numa mesma posição de um jogador, este morre.

4.2 Modelagem do Bomberman como um Processo de Decisão de Markov

Projetamos o problema do Bomberman como um MDP finito e episódico para que pudéssemos empregar os algoritmos de RL. O problema do MDP é composto pelos seguintes elementos:

- **Conjunto de estados:** é composto pelo estado de todas as células da grade no cenário e pela posição normalizada do agente (x, y). Todas as observações são passadas para o componente Brain do ML-Agents como um vetor de observação, que pode variar de acordo com o tipo de representação (detalhes a seguir). O estado terminal do agente ocorre quando ele é atingido por uma bomba ou quando é o último agente ativo da partida.
- **Ações:** O agente de aprendizagem Bomberman tem à sua disposição seis ações discretas, que são mostradas na Tabela 4.1. Todas as ações de mover, movem o agente apenas uma única célula.

Tabela 4.1: Ações Possíveis do Agente

Ação	**Descrição**
0	Ficar parado
1	Ir pra cima
2	Ir pra baixo
3	Ir pra esquerda
4	Ir pra direita
5	Colocar bomba

- **Transições:** A transição de um estado para outro, guiado por uma ação, é determinada pelo estado anterior e pela própria ação, sem nenhuma incerteza causada pelo ambiente. Assim, o agente tenta executar uma ação e, se for permitido, seu efeito é percebido pelo ambiente. Por exemplo, se ele escolher a ação de ir para cima e puder andar para cima, o próximo estado abrangerá sua nova posição. No entanto, se ele tentar executar uma ação que não é possível naquele momento, o estado permanece o mesmo (a transição é para o mesmo estado).

- **Recompensas:** As recompensas são distribuídas de acordo com a Tabela 4.2. Os valores atribuídos a cada recompensa foram configurados de forma empírica por tentativa e erro. Depois de realizar uma ação, o agente recebe uma recompensa de acordo com o novo estado do jogo. Um ou mais tipos de recompensas podem ser aplicados combinados na mesma iteração, por exemplo, o agente pode matar um inimigo e destruir um bloco *ao mesmo tempo*, ou o agente pode ficar apenas parado e sofrer a penalidade de iteração.

  Note que a recompensa oferecida ao se aproximar de um oponente (a quinta linha na tabela) é dada toda vez que um agente se aproxima ainda mais de um oponente inimigo. Por exemplo, se a distância entre os dois agentes em questão for x células, então a recompensa só será dada ao agente se ele se aproximar ainda mais do adversário, *i.e.*, se ele chegar a uma distância de pelo menos $x - 1$ células do adversário. Esse tipo de recompensa pode ser vista como atendendo a sub-objetivos do jogo, a fim de forçar os agentes a se aproximarem, de modo que um poderá matar o outro com uma bomba. Vale ressaltar que as recompensas relativas a aproximação e distanciamento (linhas 5, 6 e 7 da tabela) são dadas somente após uma ação de movimentação bem-sucedida, quando o estado observado tem uma nova posição do agente. As distâncias são calculadas como distância de Manhattan (a soma das

diferenças de cada coordenada).

Tabela 4.2: Recompensas para o Agente

Estado/Ação	**Recompensa**
O agente está morto	-0.5
Um oponente foi morto pelo agente	1.0
O agente é o último homem vivo	1.0
Um bloco foi destruído	0.1
O agente está na posição mais próxima até agora de um inimigo	0.1
O agente está mais perto de um adversário do que antes	0.002
O agente está mais longe de um adversário do que antes	-0.002
Penalidade por iteração	-0.01
Estar numa célula ao alcance de uma bomba	-0.0006666
Estar em célula segura quando há bomba por perto	0.002

4.3 Representação dos Estados

Nos algoritmos de RL, a escolha do formato de representação para os estados observados pode influenciar a capacidade de aprendizado do agente, uma vez que, se o estado for mal representado, o agente provavelmente não conseguirá encontrar padrões e realizar ações coerentes. No BLE, cada agente envia sua observação atual do ambiente para os seus respectivos Brains no ML-Agents, usando um vetor de características. O tamanho deste vetor depende do tipo de representação do estado. Por exemplo, se para representar uma célula, usamos apenas um valor numérico, o vetor de observação terá como tamanho a quantidade de células do tabuleiro multiplicada pela dimensão do valor numérico (1D); se para representar cada célula é utilizado um vetor, então o tamanho do vetor de observação será o tamanho do vetor da célula multiplicado pela quantidade de células no tabuleiro.

De acordo com a documentação do ML-Agents[53], para melhores resultados nos treinamentos, toda observação enviada para o vetor de observações deve ser normalizada na faixa de [-1, 1] ou de [0, 1]. Deste modo, a rede neural do PPO pode frequentemente convergir para uma solução de forma mais rápida. Porém não é obrigatório normalizar os valores, é apenas uma boa prática quando utiliza-se redes neurais.

No BLE, cada célula é primeiro codificada no formato de sinalizadores de bit (*bit flags*). Os sinalizadores de bits são usados para armazenar mais de um valor Booleano

em um conjunto inteiro de bytes, representando a existência de um ou mais objetos desse tipo de estado na célula. Assim, podemos informar, por exemplo, que um agente e uma bomba estão ocupando a mesma célula da grade no mesmo momento. Por exemplo, se o sinalizador de bit que representa o agente e a bomba for 01 e 10, respectivamente, o sinalizador de bit que representa a célula da grade será de 11. Existem oito tipos de estados possíveis representados por sinalizadores de bit: célula vazia, bloco indestrutível, bloco destrutível, agente atual, inimigo, bomba, fogo e perigo.

Somente, quando o agente coleta informações sobre o estado, ele pode escolher como representará as células (como adicionará a observação do ambiente no vetor de observação). Assim, a partir da codificação de sinalizadores de bits, novas formas para se visualizar (representar) as células podem ser criadas, onde deve-se converter o sinalizador de bit, que representa cada célula, para a forma de representação desejada. Nessa dissertação, criamos quatro possíveis representações para o ambiente: Flag Binária, Flag Binária Normalizada, Híbrida, ZeroOuUm. Além disso, implementamos a representação do estado introduzida em [21] e a nomeamos como ICAART (utilizamos o nome da conferência em que o artigo foi publicado). Em seguida, explicamos cada um delas.

4.3.1 Flag Binária

Representar o estado como Flag Binária é a forma mais simples de compor o vetor de observação do agente. Este vetor, além de conter a posição *(x, y)* normalizada do agente (2 observações), será preenchido com o sinalizador de bit de cada célula da grade que representa as entidades que estão em cada célula, com isso, o tamanho do vetor de observação será igual ao tamanho da grade somado com 2 observações relativas a posição do agente.

Tabela 4.3: Exemplo de Grade 2x2 para Flag Binária

	0	1
0	000	011
1	010	110

Criamos um exemplo na Tabela 4.3 para mostrar como a grade do jogo é representada utilizando-se a representação de Flag Binária. Neste exemplo, o cenário tem uma grade formada por 2 linhas e 2 colunas, ou seja, a grade possui 4 células. Neste exemplo, há 3 tipos de entidades que podem estar ocupando uma célula (ao mesmo tempo ou não): agente, indicativo de perigo e bomba, e ainda há a possibilidade de não ter nada numa

célula (vazio). Criamos um sinalizador de bit (com 3 bits) para cada entidade, o agente é representado pelo sinalizador de bit 001, o indicativo de perigo pelo 010, a bomba pelo 100, e a para representar a célula vazia apenas desativamos todos os bits do sinalizador (000). A célula da linha 0 e coluna 0 ([0,0]) é uma célula vazia, na célula [0,1] tem um agente e um indicativo de perigo, na célula [1,0] tem apenas um indicativo de perigo, e na célula [1,1] tem uma bomba e um perigo. As 2 observações relativas a posição do agente serão dadas pela posição normalizada da célula que tem o agente, ou seja, a posição [0,1].

Cada sinalizador de bit relativo a cada célula é convertido num valor decimal (conversão de número binário para decimal) ao ser enviado para o vetor de observação, por exemplo, o sinalizador de bit da célula [1,0] que é 010 será convertido no decimal 2. O tamanho do vetor de observação do agente será a soma do tamanho da grade (linha × coluna) do tabuleiro somado com as 2 observações relativas à posição (x, y) do agente. Para o exemplo acima, o tamanho do vetor de observação é igual ao tamanho da grade que é 4 somado com as 2 observações de posição, resultando num total de 6 observações.

Dois ou mais estados podem ser representados ao mesmo tempo. Se em uma célula está presente o agente (001) e a bomba (100), então podemos combinar os sinalizadores 001 | 100 resultando em 101. Note que esse valor binário é convertido em decimal, resultando em 1 + 4 = 5. Tais valores decimais é que são repassados para o vetor de observação do agente.

4.3.2 Flag Binária Normalizada

A representação como Flag Binária Normalizada apenas normaliza a representação anterior, de Flag binária, das células de acordo com os valores máximos e mínimos permitidos na representação de Flag Binária. Os valores são números em ponto flutuante na faixa entre 0 e 1. Com a normalização, pretendemos obter agentes mais inteligentes comparados com os agentes que não utilizam normalização, porque como diz na documentação do ML-Agents[53], redes neurais convergem mais rápido quando os valores observados são normalizados.

Tabela 4.4: Exemplo de Grade 2x2 para Flag Binária Normalizada

	0	1
0	0,0	0,43
1	0.29	0,86

No exemplo da Tabela 4.4, o menor valor possível para representar uma célula com a representação de Flag Binária é 0, e o maior valor possível é 7. Neste exemplo, o cenário é o mesmo da Tabela 4.3, com as mesmas entidades em cada célula. Normalizamos o sinalizador de bit presente em cada célula utilizando os valores mínimos e máximos informados anteriormente. A célula da linha 0 e coluna 0 ([0,0]) é uma célula vazia e seu valor normalizado é 0.0, na célula [0,1] tem um agente e um indicativo de perigo resultando no valor binário 011 que é convertido para decimal (011 $\rightarrow$ 3) e normalizado (3/7) resultando em 0,43, na célula [1,0] tem apenas um indicativo de perigo (010 $\rightarrow$ 4) que normalizado vira 0,29, e na célula [1,1] tem uma bomba e um perigo (110 $\rightarrow$ 6) que normalizado é 0,86. As 2 observações relativas à posição do agente continua sendo a posição normalizada da célula que tem o agente, ou seja, a posição [0,1].

Cada decimal da representação de Flag Binária é normalizado resultando num número de ponto flutuante entre 0 e 1 (0 <= i <= 1) ao ser enviado para o vetor de observação. Por exemplo, o sinalizador de bit da célula [1,0] que é 010 será convertido no decimal 2 e normalizado (2 / 7), resultando no valor 0,29. O tamanho do vetor de observação deste tipo de representação será igual ao de Flag Binária. Note que a única diferença é que os valores que representam as células são normalizados.

Dois ou mais estados podem ser representados ao mesmo tempo. Se em uma célula está presente o agente e a bomba, então os sinalizadores podem ser combinados (001 | 100) / 111 = 101/111 = 0,714. Note que os valores binários, que são convertidos em decimais e depois normalizados é que são repassados para o vetor de observação do agente.

4.3.3 Híbrida

A representação Híbrida combina os conceitos da representação de Flag Binária com uma representação de vetor *One-Hot*. Um vetor *One-Hot* é uma matriz $1 \times C$, em que C é o número de situações possíveis, que consiste em números 0 em todas as dimensões, com exceção de um único 1 em uma dimensão, usado exclusivamente para identificar o tipo de classe ou estado. O tipo de representação de vetor *One-Hot* é o tipo de representação recomendado pelo ML-Agents para representar tipos enumerados de dados [53]. Por exemplo, para representar uma célula que tenha um agente (vetor *One-Hot* do agente [1,0]) e uma bomba (vetor *One-Hot* da bomba [0,1]) com um vetor *One-Hot*, a dimensão do vetor deve ser aumentada em um, gerando um novo vetor *One-Hot* que represente o agente mais a bomba ([0,0,1]). Essa característica pode fazer com que tais vetores fiquem com um tamanho excessivo. Por causa disso, propomos uma representação híbrida que

permita usar um vetor *One-Hot* similar à representação de Flag Binária, em que não ficamos presos no conceito de ativar somente uma dimensão do vetor com o valor 1. Ou seja, para representar uma bomba ([0,1]) e um agente ([1,0]) ocupando a mesma célula da grade, temos o vetor Híbrido [1,1] em vez de um novo vetor *One-Hot* [0,0,1]. Cada célula na grade é representada usando um vetor Híbrido. Assim, se houver 4 tipos de estado (mais vazio), o tamanho de cada vetor Híbrido será 4 (por exemplo, [1,0,0,1]) e, consequentemente, o tamanho do vetor de observação será o número de células da grade, multiplicado pelo tamanho do vector Híbrido mais a posição x, y (por exemplo, $4 \times 4 + 2 = 18$, em uma grade de 2×2).

De uma forma mais matemática, um vetor Híbrido é uma matriz $1 \times H$, em que H é o número de situações possíveis, que consiste em números 0 em todas as dimensões, com exceção de números 1 em cada dimensão (tipo de classe) quando seu respectivo estado está ativo.

Tabela 4.5: Exemplo de Grade 2x2 para representação Híbrida

	0	1
0	[0, 0, 0]	[1, 1, 0]
1	[0, 1, 0]	[0, 1, 1]

A Tabela 4.5 apresenta a mesma grade 2×2 contendo 4 células mostrada nos exemplos das outras representações anteriores, porém utiliza a representação Híbrida para representar cada célula. Há 3 tipos de entidades que podem ocupar uma célula (ao mesmo tempo ou não). O agente é representado pelo vetor híbrido ([1, 0, 0]), o indicativo de perigo pelo vetor híbrido ([0, 1, 0]), a bomba pelo vetor híbrido ([0, 0, 1]), e a célula vazia é representada pelo vetor híbrido ([0, 0, 0]). A célula [0,0] é uma célula vazia, na célula [0,1] tem um agente e um indicativo de perigo, na célula [1,0] tem apenas um indicativo de perigo, e na célula [1,1] tem uma bomba e um perigo. As 2 observações relativas a posição do agente continuam sendo a posição normalizada da célula que tem o agente, ou seja, a posição [0,1].

Cada vetor híbrido relativo a cada célula é adicionado ao vetor de observação. Por exemplo, o vetor híbrido da célula [1,0] que é ([0, 1, 0]) é adicionado em sequência ao vetor de observação como 3 novas observações. Esses valores são usados como número de ponto flutuante. O tamanho do vetor de observação do agente será a soma do tamanho do vetor híbrido multiplicado pelo tamanho da grade do tabuleiro do jogo, e somado com a posição normalizada (x, y) do agente. Para o exemplo acima, o tamanho do vetor de

observação é igual ao tamanho do vetor híbrido que é 3 multiplicado pelo tamanho da grade que é 4 e somado com as 2 observações de posição, resultando num total de 14 observações.

Podemos também representar dois ou mais estados ao mesmo tempo. Se em uma célula está presente o Agente e Bomba, então podemos combinar os vetores híbridos: [0,0,1] + [1,0,0] = [1,0,1]. Note que cada valor do vetor híbrido é um número de ponto flutuante. Quanto maior o número de entidades a serem representadas utilizando um vetor híbrido, maior será o número de observações, pois cada célula da grade precisará ser representada por um vetor desses.

4.3.4 ICAART

Essa é a representação do estado utilizada em [21]. Primeiro, para cada célula da grade é adicionado ao vetor de observação do agente o valor 1, se a célula está Livre ou 0, se ela tem um bloco destrutível, ou -1, se está obstruída (bloco indestrutível ou bomba). Depois, para cada célula da grade é adicionado ao vetor de observação o valor 1, caso o agente esteja nessa posição ou 0, caso contrário. Após isso, para cada célula da grade, é adicionado o valor 1, caso haja oponentes ou 0, caso contrário. E, por último, para cada célula do grade, é adicionado ao vetor de observação o nível de perigo de uma posição (-1,0 <= perigo <= 1,0). O perigo é medido de acordo com o tempo transcorrido, dividido pelo tempo necessário para a bomba explodir, onde uma bomba leva N iterações de tempo depois que ela é colocada até que ela exploda. O valor de perigo é negativo se a bomba foi colocada pelo agente e positiva se foi colocada por um adversário ou ambiente. Por essa representação utilizar o nível de perigo de uma célula, ela passa a usar uma informação privilegiada. Diferente de todos os outros tipos de representação criados por nós.

Nesta representação, o tamanho do vetor de observação do agente é 4 vezes maior que o tamanho da grade, porque como foi explicado anteriormente, itera-se pela lista de células da grade uma vez para se decidir se a célula está livre, obstruída por bloco destrutível ou indestrutível, itera-se outra vez para informar onde o agente está, itera-se outra vez para informar onde os inimigos estão, e por último itera-se mais uma vez para informar o nível de perigo de cada célula.

Para o exemplo de grade da Tabela 4.3, considerando que a bomba foi colocada pelo próprio agente, e que ela leva 6 iterações para explodir e que faltam 3 iterações para que isto ocorra, o vetor de observação resultante do agente ficaria assim: [1,1,1,-1, 0,1,0,0, 0,0,0,0, 0,0.5,0.5, 0.5]. Ou seja, para cada célula da grade verificamos se está livre, obs-

truída por bloco destrutível, ou por indestrutível ou bomba e geramos e adicionamos os valores no vetor de observação (1,1,1,-1). Depois verificamos cada célula e informamos 1 na célula onde o agente está (0,1,0,0) e adicionamos ao vetor de observação. Em sequência, informamos os inimigos e adicionamos ao vetor de observação (0,0,0,0). E por fim calculamos o nível de perigo de cada célula, a célula [0,0] não é atingível pela bomba da célula [1,1] então seu nível de perigo é 0, as células [0,1], [1,0] e [1,1] tem nível de perigo e como a bomba foi colocada a 3 iterações e leva 6 para explodir, calculamos o nível de perigo dividindo 3 por 6 e resultando no nível de perigo de 0,5. Portanto os valores adicionados ao vetor de observação relativos ao nível de perigo são (0,0.5,0.5,0.5).

Com representação ICAART é possível representar uma célula da grade que possua dois ou mais estados também, porém de uma forma diferente. Note que as observações foram agrupadas por tipo de observação nessa implementação, isto é, itera-se por cada célula da grade por 4 vezes gerando sequências de observações por lote. Entretanto, acreditamos que se estas observações fossem agrupadas por célula, como na representação Híbrida (poderíamos iterar uma única vez e fazer os 4 testes), provavelmente não haverá grandes mudanças no modelo que é gerado após o treinamento utilizando este tipo de representação. Agrupar por célula, quer dizer por exemplo, que para cada célula seriam testados todas as verificações de uma vez.

4.3.5 ZeroOuUm para cada tipo de célula

Esse tipo de representação se assemelha muito à representação Híbrida, porém ao invés de adicionar um vetor híbrido para cada célula da grade, adiciona-se ao vetor de observação cada observação separadamente, como foi feito na representação do ICAART. Segue-se a ordem para cada sequência de iteração na lista de células da grade:

1. Célula livre ou obstruída (1 ou 0 respectivamente)
2. Célula destrutível ou não (1 ou 0 respectivamente)
3. Célula contem o agente (1 ou 0 respectivamente)
4. Célula possui um oponente (1 ou 0 respectivamente)
5. Célula com bomba (1 ou 0 respectivamente)
6. Célula com perigo (1 ou 0 respectivamente)
7. Célula com fogo (1 ou 0 respectivamente)

Nesta representação, o tamanho do vetor de observação do agente é 7 vezes maior que o tamanho da grade, porque itera-se pela lista de células da grade uma vez para cada item relatado acima. Para o exemplo de grade da Tabela 4.3, o vetor de observação resultante ficaria assim: [1,1,1,0, 0,0,0,0, 0,1,0,0, 0,0,0,0, 0,0,0,1 0,1,1,1 0,0,0,0]. Cada 4 observações deste vetor é relativa a um item relatado acima.

Com a representação ZeroOuUm é possível representar uma célula da grade que possua dois ou mais estados, porém de uma forma diferente. Esta representação pode ser vista como uma espécie de representação Híbrida sem a necessidade de se agrupar as informações de cada célula por vez. Ao invés disso, as informações são agrupadas por tipo de observação. Note também que, nesse caso, o estado livre é representado por 1, ao invés de 0 como nos nas representações de Flag Binária e Flag Binária Normalizada.

4.4 Algoritmos de Aprendizagem por Reforço

O BLE inclui três métodos para fazer o agente aprender: (i) usando o algoritmo de aprendizagem por reforço PPO [45]; (ii) usando o algoritmo BC [25]; e (iii) usando primeiro o BC, seguido pelo aprendizado com PPO, mas esse último começando com o modelo aprendido por BC. Em todos esses casos, permitimos o uso de qualquer tipo de representação dos descritos anteriormente. Além de possibilitar o uso de uma MLP, o BLE também permite o uso de uma rede de memória de curto prazo recorrente [14] como o método de aproximação de funções dentro do processo de aprendizado. Com isso, o agente pode considerar suas experiências passadas para induzir a política. Todos esses algoritmos são implementados no ML-Agents.

O algoritmo PPO segue a técnica actor-critic A3C [29], permitindo que múltiplos agentes assíncronos possam ser treinados simultaneamente em uma rede de supervisor global [23]. Na implementação do PPO do ML-Agents, isso é feito usando um ou mais Brains. Cada Brain é responsável por um ou mais agentes, sendo que eles recebem como entrada a observação de seus respectivos agentes, e retornam como saída as ações a serem tomadas pelos agentes. Durante cada iteração do jogo, o método acumula em uma estrutura de dados a experiência de cada agente composta do estado atual, da ação executada, da recompensa recebida e do próximo estado. Após um determinado número de experiências coletadas, a função de valor e a função Vantagem são executadas para que a política possa ser atualizada. Note que cada Brain treina a política do agente apenas com as experiências de seus respectivos agentes.

Para treinar o agente usando o algoritmo BC, fornecemos um arquivo de repetição com as demonstrações de um especialista[3]. O número de oponentes em cada partida pode mudar de acordo com o arquivo de repetição. A partida em que o agente especialista jogou é reproduzida exatamente como no arquivo de repetição para que, com o tempo, o especialista possa demonstrar ao aluno como jogar o jogo. Quando o aluno está aprendendo, todos os movimentos dos oponentes são sincronizados com o arquivo de repetição. Neste caso, seus oponentes reagem de acordo com o arquivo de repetição, mas o estudante toma suas próprias decisões. Depois que o agente aluno morre ou o número de iterações por partida atinge um limite máximo, a próxima partida no arquivo é carregada.

Também permitimos combinar o treinamento de PPO e BC, realizando primeiramente o treinamento com o método BC, seguido pelo treinamento com PPO (Veja o Algoritmo 1). Na sequencia, salva-se o modelo treinado com o BC e carrega-se esse modelo para inicializar os pesos da rede PPO. Usamos uma função que força a rede a ser carregada, mesmo que a rede utilizada pelo BC e a rede utilizada pelo PPO tenham alguma diferença em relação ao número e configurações de camadas, porém tentamos configurar as duas redes da forma mais semelhante possível. Para tanto, cada neurônio de cada rede neural recebe um identificador. Então, todos os neurônios na rede neural corrente que possuem o mesmo nome que na rede neural anterior são carregados com os valores do treinamento anterior, enquanto todos os nós que não estavam presentes no treinamento anterior são inicializados com seus valores padrões. Mesmo que a arquitetura principal de ambos os métodos seja similar, no que diz respeito à quantidade de camadas ocultas e neurônios em cada camada, a rede do PPO terá ao menos camadas de saída adicionais, por exemplo, para calcular a função de valor.

Algoritmo 1 Treinamento BC mais PPO

- Gravar partidas em um arquivo de um jogador especialista contra agentes oponentes.{Cenários devem ser diversificados e quantidade de oponentes também para um treinamento bem sucedido.}
- Executar treinamento do BC por M iterações utilizando o arquivo de repetições para demonstrar os movimentos do especialista ao aprendiz, e gerar um modelo.
- Iniciar treinamento do PPO a partir do modelo do BC.{Os pesos da NN são iniciados de acordo com o modelo BC}
- Continuar o treinamento do PPO por N iterações.

Na Figura 4.1, mostramos em resumo a interseção entre as redes neurais do BC e do

[3]No caso, o especialista foi o próprio autor desta dissertação

PPO dentro do BLE. Como podemos ver, a rede neural do BC está contida na rede neural do PPO. A primeira camada da rede é a camada de entrada, seu tamanho vai depender do tamanho do vetor de observação, que varia de acordo com o tipo de representação de estado escolhido. Em seguida, temos 3 camadas escondidas que possuem 128 neurônios cada, interligados densamente. A camada de saída, que é a interseção do BC com PPO, possui tamanho 6 porque no BLE há 6 ações possíveis (parado, esquerda, direita, cima, baixo e colocar bomba). Esta camada informa a probabilidade de cada ação ser escolhida de acordo com as observações que entraram na rede. O BC atualiza essas probabilidades de acordo com uma função de perda que compara as ações que são demonstradas pelo especialista com a ação de maior probabilidade da camada de saída. Já o PPO, calcula uma função de valor como saída e faz um passo que também atualiza as probabilidades das ações de acordo como foi explicado na Seção 2.2.4. Resumidamente, no treinamento posterior com o PPO, nós inicializamos cada neurônio das camadas escondidas e das probabilidades das ações com os valores treinados no treinamento prévio do BC.

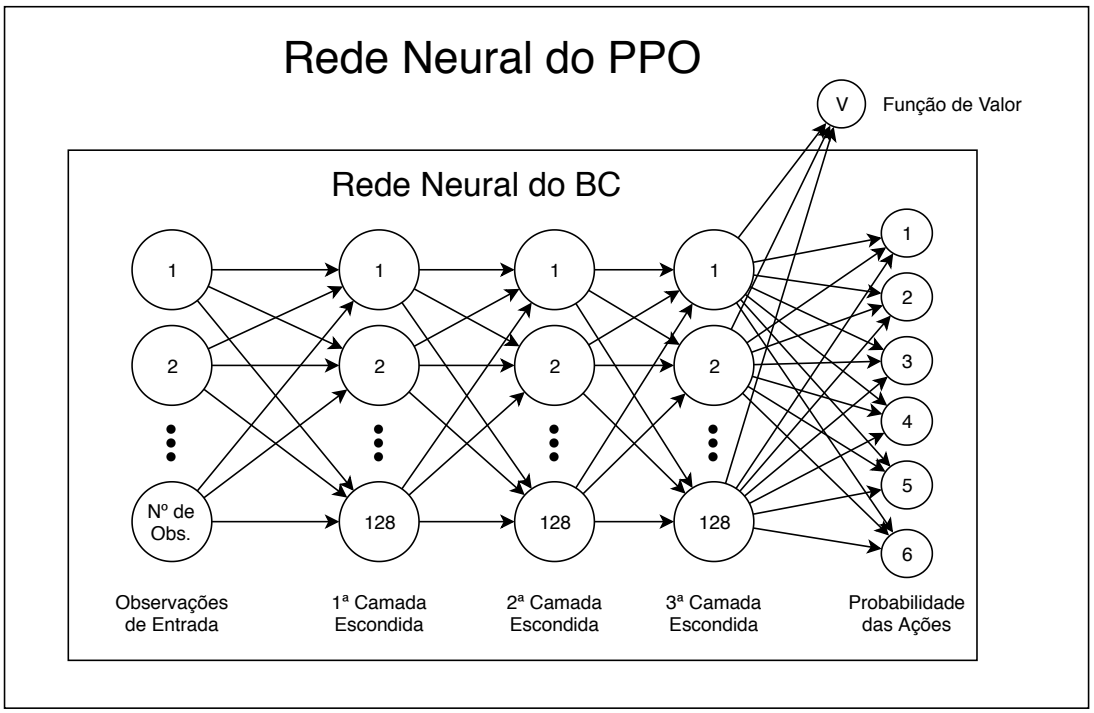

Figura 4.1: Interseção resumida entre as redes neurais do BC e do PPO dentro do BLE

4.5 Processo de Treinamento

Todo o processo de treinamento é composto de várias partidas correspondentes aos episódios de um algoritmo tradicional de RL. No início de cada partida, os agentes são criados

e posicionados aleatoriamente na grade. Uma partida termina quando não há agentes vivos. Como há mais de um agente por cenário, é necessário sincronizar as observações, ações e recompensas para que um agente não tire proveito de outro devido a ter obtido informações antes dos outros agentes. Se houver pelo menos um agente vivo, o BLE observa as ações que serão tomadas pelos agentes. Antes de realizar uma ação é necessário atualizar o estado das bombas no cenário, para certificar-se que é o momento correto para explodi-las e para que o efeito de tais explosões sejam adequados. Também é necessário atualizar os blocos, pois, alguns deles, existentes na iteração anterior, podem ter sido atingidos por uma explosão. Em seguida, cada agente recebe uma observação atualizada, representando o estado atual do ambiente, e a recompensa correspondente ao estado observado. Após atualizar a observação, o BLE envia ao Brain as observações do estado atual do ambiente e as recompensas relativas da iteração anterior. Depois disso, a função de política é atualizada se já acumulou uma quantidade X de experiências. Finalmente, se o agente ainda estiver ativo e não mais no estado Finalizado, ele irá atuar no ambiente, de acordo com a ação calculada pela função de política atual que é retornada pelo Brain. No final de uma iteração, a recompensa de penalidade de tempo é aplicada.

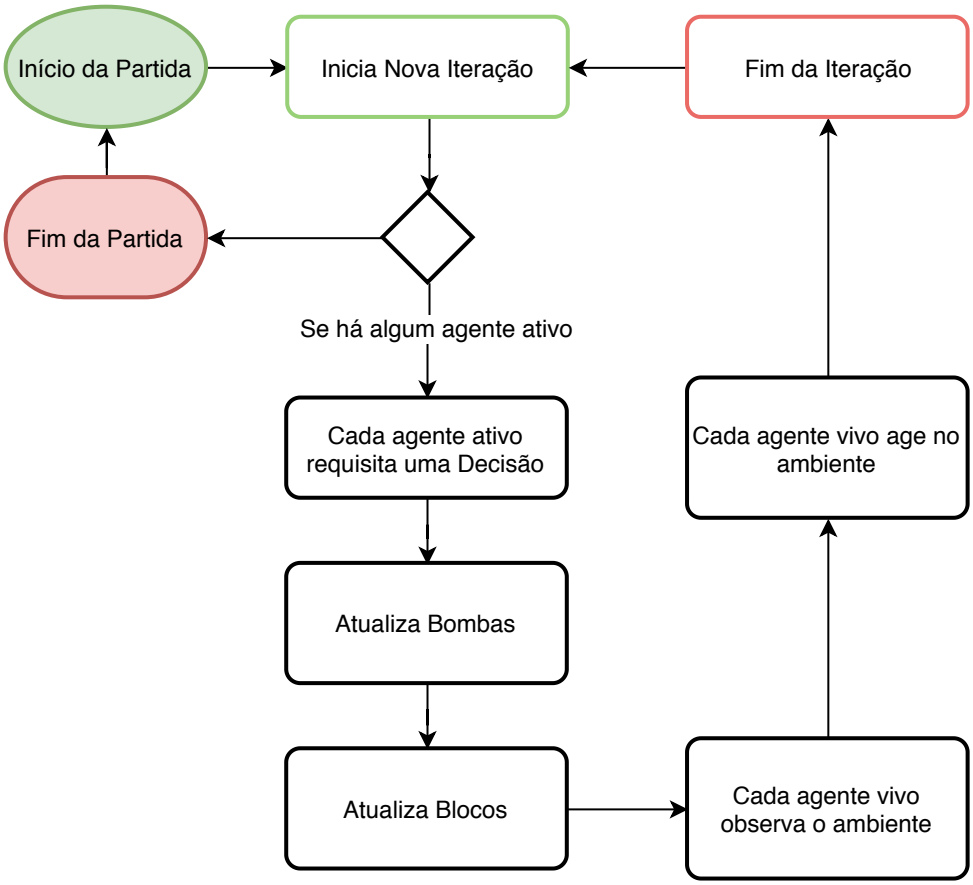

Figura 4.2: Diagrama do fluxo do Processo de Treinamento do BLE

Capítulo 5

Resultados e Análises

No BLE, podemos utilizar diferentes formas de representação do estado do jogo, podemos treinar um ou vários Brains simultaneamente, podemos utilizar diferentes algoritmos para treinar os agentes, tais como PPO, PPO com LSTM, BC, BC seguido de PPO ou desenvolver qualquer outro algoritmo que esteja implementado de acordo como é esperado pelo kit de ferramentes ML-Agents. Neste capítulo, temos como objetivo investigar empiricamente: (1) qual é a melhor forma de se representar o estado, (2) qual treinamento entre Um-Brain e Multi-Brain é melhor, (3) se utilizar LSTM com PPO melhora o desempenho dos agentes treinados, e (4) se um treinamento prévio com BC influencia positivamente um treinamento posterior com o PPO. Para tanto, neste capítulo, apresentamos a configuração experimental, os resultados e as análises realizadas com o BLE. Primeiro, relatamos as especificações técnicas para a execução dos experimentos na Seção 5.1.1. Na Seção 5.1.2, informamos todos os parâmetros de configuração do BLE que foram utilizados nos treinamentos e torneios. Por fim, explicamos e apresentamos os experimentos individualmente, e esses estão enumerados abaixo:

1. Comparação entre os 5 tipos de representação de estado presentes no BLE, apresentadas na Seção 4.3: realizamos este experimento para descobrirmos qual é a melhor forma de se representar o estado da grade do jogo ao enviar estas informações para o vetor de observações do algoritmo PPO. Para este experimento, realizamos treinamentos com agentes Um-Brain para cada tipo de representação de estado. Após os treinamentos, realizamos uma série de torneios em que os agentes batalham entre si com o objetivo de verificar qual é o mais vitorioso. Consequentemente, o tipo de representação de estado que o agente vitorioso utilizou é considerada a melhor representação de estado dentre as 5 apresentadas (Seção 5.2).

2. Comparação do treinamento Multi-Brain com o treinamento Um-Brain, conforme discutido na Seção 2.3: comparamos se o treinamento Multi-Brain gera agentes mais vitoriosos que o treinamento One-Brain. Optamos por não utilizar apenas a representação Híbrida, que foi a representação vitoriosa do experimento anterior, para confirmarmos se os resultados quanto ao melhor tipo de representação iria se alterar neste treinamento Multi-Brain. Neste treinamento, cada Brain é responsável por um agente em cada um dos cenários presentes no BLE, e cada Brain utiliza um tipo de representação de estado diferente. Após os treinamentos, fizemos torneios entre estes Brains para encontrar o mais vitorioso. Também fizemos outros torneios entre os primeiro e segundo colocados deste experimento e os primeiro e segundo colocados do primeiro experimento. Ou seja, comparamos se o treinamento Multi-Brain é melhor do que o treinamento Um-Brain e se o melhor tipo de representação iria se alterar (Seção 5.3).

3. Comparação entre os métodos PPO e PPO com LSTM: todos os experimentos anteriores utilizam o algoritmo PPO com MLP. Nesse experimento, realizamos 5 treinamentos Um-Brain que utilizaram o método PPO com LSTM e a representação de estado Híbrida (representação vitoriosa nos treinamentos anteriores). Após o treinamento, fizemos torneios entre os Brains que utilizaram os modelos gerados neste treinamento para descobrirmos qual modelo entre eles era o mais vitorioso, e depois colocamos este modelo para batalhar com o Brain vitorioso do primeiro experimento (Um-Brain PPO com representação Híbrida) (Seção 5.4).

4. Comparações entre o método BC+PPO e os demais Brains vitoriosos: realizamos um treinamento prévio com o método BC e depois continuamos o treinamento com o método PPO. Após o treinamento, fizemos torneios para analisar se esta nova forma de treinar é mais vitoriosa que as demais, e se é vantajoso fazer esse treinamento prévio com o BC (Seção 5.5).

A Figura 5.1, mostra resumidamente o fluxo básico dos experimentos realizados nesta dissertação. Primeiro, realizamos os treinamentos para gerar os modelos, depois realizamos torneios para descobrir quais são os modelos mais vitoriosos, e por fim, analisamos os resultados dos torneios.

Figura 5.1: Fluxo Básico dos Experimentos Realizados.

Na Tabela 5.1 colocamos os ponteiros para todas as tabelas, gráficos e fluxogramas que foram criados para cada experimento.

Tabela 5.1: Experimentos realizados no BLE

Nome	Tabelas	Gráficos	Fluxogramas
Um-Brain quanto a Melhor Representação	5.4, 5.5, 5.6, 5.7, 5.8, 5.9, 5.10, 5.11	5.4, C.1, C.2, C.3, C.4, C.5, C.6, C.7	5.3
Multi-Brain	5.12, 5.13, 5.14, 5.15, 5.16, 5.17	5.6, C.8, C.9, C.10, C.11, C.12	5.5, 5.7
Um-Brain LSTM	5.18, 5.19, 5.20	C.13, C.14, C.15, C.16, C.17, C.18	5.8
BC+PPO	5.21, 5.22, 5.23, 5.24, 5.25, 5.26, 5.27, 5.28, 5.29	-	5.9

Todas as estatísticas e todos modelos gerados nos treinamentos dos agentes no BLE estão disponíveis[1]. Todos os gráficos de linha contendo estas estatísticas dos treinamentos foram suavizados com um passo de 100K iterações por registro. Abaixo, adicionamos uma descrição resumida para cada estatística gerada pelos treinamentos com o ML-Agents[2].

1. **Recompensa Acumulada:** é a recompensa média do episódio sobre todos os agentes de um determinado Brain. Deve aumentar durante uma sessão de treinamento bem sucedida.
2. **Entropia:** quão aleatórias são as decisões do modelo. Deve diminuir lentamente durante um processo de treinamento bem sucedido. Se diminuir muito rapidamente, o hiper-parâmetro beta deve ser aumentado. Beta é a força da regularização de entropia, o que faz a política mais aleatória.

[1] https://drive.google.com/file/d/14LEpfMvDez_ILnblMtKBIQH9RgDiyYzb/view?usp=sharing

[2] Explicçaão extraída da documentação do ML-Agents https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Using-Tensorboard.md

3. **Tamanho do episódio:** é a duração média em passos de simulação (iteração) de cada episódio no ambiente para todos os agentes de um determinado Brain.
4. **Loss de Política:** é a magnitude média da função de perda de política. Está correlacionada com o quanto a política está mudando. A magnitude deve diminuir durante uma sessão de treinamento bem-sucedida.
5. **Estimativa de Valor:** é a estimativa do valor médio para todos os estados visitados pelo agente de um determinado Brain. Deve aumentar durante uma sessão de treinamento bem sucedida.
6. **Loss de valor:** é a perda média da atualização da função de valor. Está correlacionada ao quão bem o modelo é capaz de prever o valor de cada estado. Essa estatística deve aumentar enquanto os agentes de determinado Brain estiverem aprendendo e, em seguida, diminuir quando a recompensa se estabilizar.

5.1 Metodologia Experimental

Nessa seção, detalhamos as especificações do hardware utilizado para a execução dos experimentos e os parâmetros utilizados para os diferentes experimentos.

5.1.1 Especificações Técnicas

A máquina utilizada para os treinamentos do BLE foi a DGX do Instituto de Computação da UFF (Veja a Tabela 5.2 que contem as especificações). Apesar de ser uma máquina com várias GPUs, os experimentos realizados com TensorFlow GPU não se mostraram mais rápidos e portanto optamos por utilizar a versão do TensorFlow CPU que utilizava menos memória RAM nos treinamentos. Criamos duas imagens de docker nVidia: uma contendo a configuração do pacote python do ML-Agents para rodar com TensorFlow GPU, e outra contendo a configuração para rodar o TensorFlow CPU. As duas imagens podem ser baixadas no site hub.docker.com[3].

[3]https://hub.docker.com/r/icaro56/ml-agents_images/tags/

Tabela 5.2: Detalhes das Especificações da máquina DGX do Instituto de Computação da Universidade Federal Fluminense

Especificações do Sistema	
GPUs	8x Tesla GP100
TFLOPS (GPU FP16 / CPU FP 32)	170/3
Memória GPU	16 GB por GPU
CPU	Dual 20-core Intel Xeon E5-2698 v4 2.2 GHz
Núcleos de NVIDIA CUDA	28672
Memória RAM do Sistema	512 GB 2133 MHz DDR4 LRDIMM
Armazenamento	4x 1.92 TB SSD RAID 0
Rede	Dual 10 GbE, 4 IB EDR
Softwares	Ubuntu Server Linux OS Controlador de GPU Recomendado DGX-1

5.1.2 Parâmetros gerais dos experimentos

Definimos o número de iterações para explodir uma bomba como $N = 6$, enquanto o fogo da bomba é definido para durar apenas 1 iteração e tem um alcance de duas células para cada direção. Todas as grades dos cenários são compostas de 9 linhas por 9 colunas. No início de uma partida, podemos criar de forma aleatória de 2 a 4 agentes por cenário, o que é feito no treinamento Um-Brain (PPO, e PPO com LSTM). Já no treinamento Multi-Brain, criamos 4 agentes por cenário de modo fixo. As especificidades dos experimentos com o BC são explicadas na sua seção específica. Para treinar os agentes, criamos e configuramos 10 cenários que executam em paralelo no mesmo ambiente 3D, simulando diferentes episódios do algoritmo RL. Entre eles, cinco são estáticos no que diz respeito aos blocos destrutíveis, o que significa que estes são recriados nas mesmas posições no início de cada episódio, e os outros cinco são configurados aleatoriamente, ou seja, no início de cada episódio para cada posição em que pode existir um bloco destrutível, sorteamos aleatoriamente se ali existirá um bloco destrutível ou não. A Figura 5.2 mostra cinco cenários estáticos e um aleatório. O cenário aleatório é o cenário da segunda linha e terceira coluna.

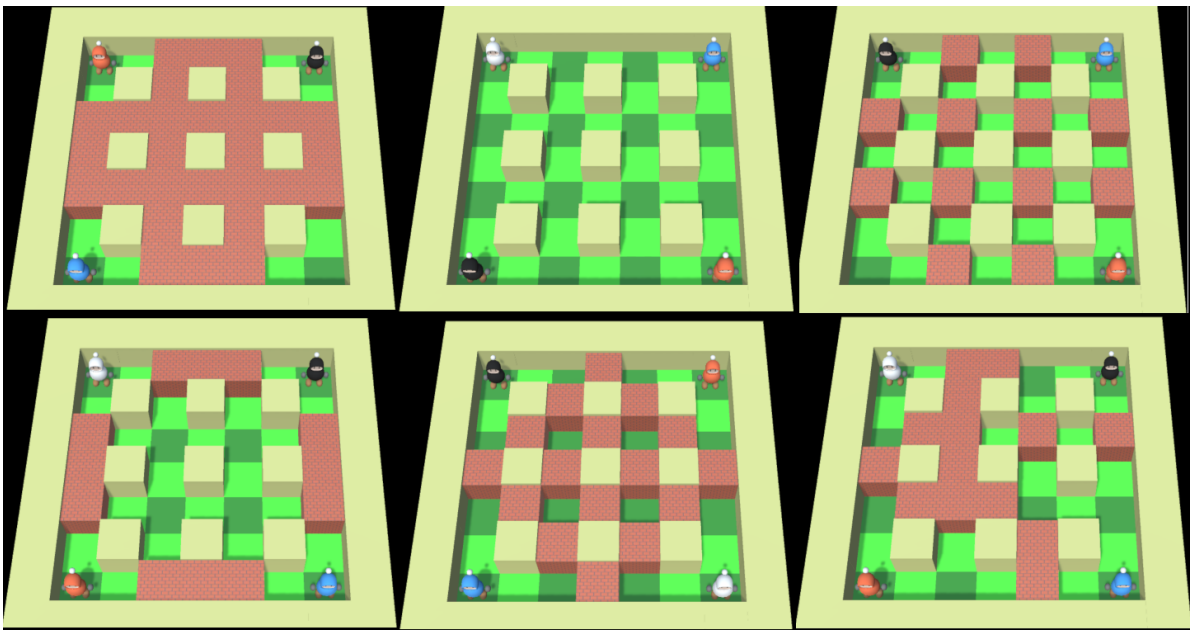

Figura 5.2: Exemplos de cenários estáticos e um aleatório (segunda linha, terceira coluna) com grade 9x9 do BLE.

Criamos dois tipos de torneios para os agentes/Brains batalharem entre si, sendo um torneio composto por 100 partidas (T100) e outro por 1000 partidas (T1000). Note que essa contagem de partidas é individual para cada cenário: o BLE geralmente traz dez cenários, ou seja, num T100 cem partidas são disputadas em cada um dos dez cenários. Os parâmetros dos treinamentos e torneios que são diferentes entre os experimentos foram colocados na Tabela 5.3 para facilitar e resumir o entendimento das configurações. Para a execução de cada um dos métodos, determinamos os hiper-parâmetros por tentativa e erro, de forma empírica.

Tabela 5.3: Parâmetros do Jogo configurados para cada experimento

	Treinamentos				**Torneios**	
Nome do Experimento	controlando 2 a 4 agentes por cenário	controlando de modo fixo 4 agentes por cenário	controlando apenas um agente por cenário	controlando apenas um agente num único cenário que é carregado via arquivo de repetição (o cenário se altera)	Torneio com 4 agentes, cada um por si	Torneio com 2 agentes, cada um por si
Um-Brain quanto a Melhor Representação	***X***				***X***	
Multi-Brain		***X***			***X***	
Um-Brain LSTM	***X***				***X***	
Experimentos realizados na Seção do BC+PPO						
BC				***X***		***X***
BC+PPO_Only1			***X***			***X***
BC+PPO_All	***X***					***X***
PPO_Only1			***X***			***X***

5.2 Escolhendo a representação de estado para o jogo do Bomberman

Resumidamente, todo experimento realizado nesta seção é mostrado na Figura 5.3 por meio de um fluxograma. Primeiro realizamos os treinamento Um-Brain, gerando 5 modelos para cada tipo de representação. Depois removemos o modelo com pior recompensa acumulada de cada tipo de representação gerando novos conjuntos para a realização de torneios prévios (T100). Com os modelos vencedores de cada T100, criamos um novo torneio T1000 para os modelos batalharem entre si, porém primeiro tivemos que colocar para batalhar num T100 os 2 modelos que obtiveram os piores rendimentos durante o treinamento (FB e FBN) para que eles disputassem a última vaga. O modelo vencedor do T1000 foi o H_1.

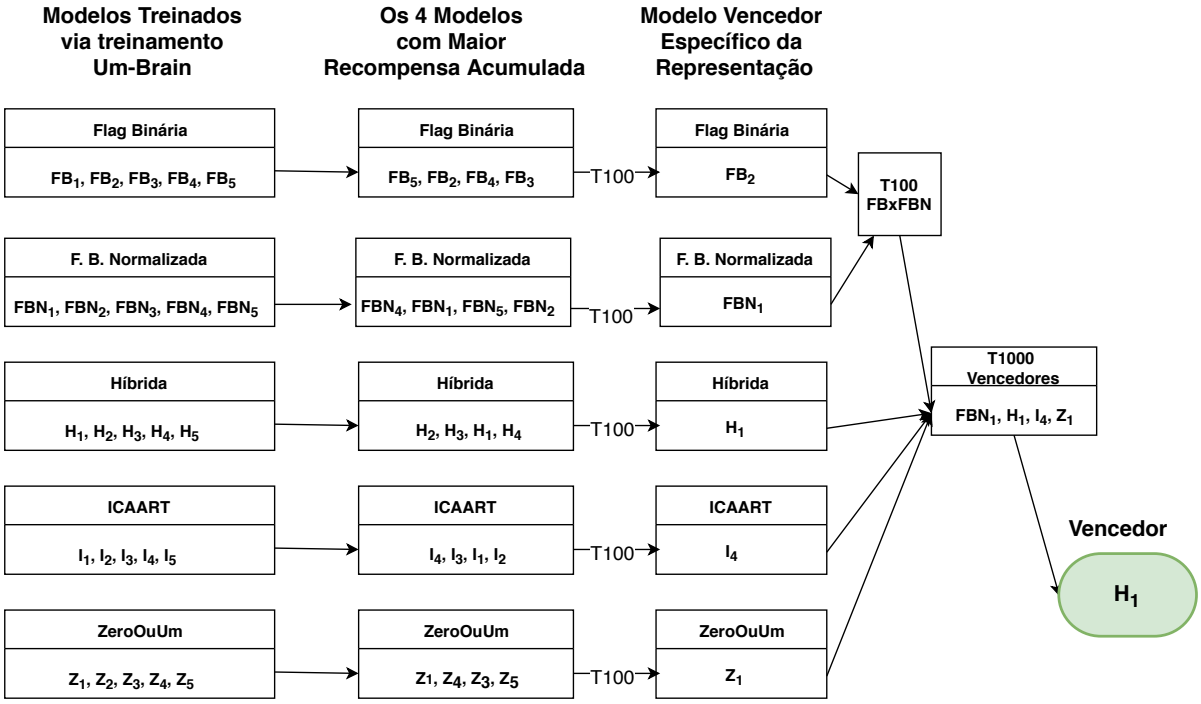

Figura 5.3: Fluxo do experimento para descobrir qual a melhor representação de estado para o jogo Bomberman

De modo a selecionar a forma mais apropriada de representar os estados no BLE, treinamos o PPO com com a configuração de um Brain do ML-Agents (Um-Brain), ou seja, aprendendo com a experiência de todos os agentes de cada um dos 10 cenários. Repetimos o treinamento cinco vezes por 2M iterações para cada um dos cinco tipos de representação de estado (Flag Binária, Flag Binária Normalizada, Híbrida, ICAART e ZeroOuUm). Assim, temos um conjunto *homogêneo_brain* = $\{\boldsymbol{FB}, \boldsymbol{FBN}, \boldsymbol{H}, \boldsymbol{I}, \boldsymbol{ZU}\}$, com cada elemento no conjunto correspondente a uma das 5 representações de estado. Cada elemento em *homogêneo_brain* se desdobra em cinco conjuntos de treinamento, *i.e.*, $\boldsymbol{FB} = \{FB_1, \ldots, FB_5\ \}$, $\boldsymbol{FBN} = \{FBN_1, \ldots, FBN_5\}$, onde o primeiro elemento corresponde a uma execução de treinamento inteira, o segundo elemento corresponde a outra execução de treinamento independente, e assim por diante. Assim, temos um total de 25 execuções de treinamento. Determinamos todos os hiper-parâmetros por tentativa e erro, de forma empírica, para treinar os Brains com PPO, e eles estão listados na Tabela 5.4. Em média, foram necessárias 20 horas para que cada treinamento fosse concluído. Durante o treinamento, observamos que quanto maior o vetor de observação, mais tempo o treinamento leva para terminar, ou seja, as representações Híbrida e ZeroOuUm levem mais tempo para treinar. No Apêndice A há uma descrição detalhada de cada hiper-parâmetro. O hiper-parâmetro max steps configura o número máximo de iterações do jogo, conside-

rando todas as iterações independentemente do episódio (partida). Assim, um episódio pode possuir mais iterações do que outro, porém toda iteração demora o mesmo tempo de jogo, ou seja, uma iteração é igual o seguinte fluxo: agentes observam, depois agentes agem.

Tabela 5.4: Hiper-parâmetros do PPO

Nome	**Valor**
batch size	128
num layers	3
hidden units	128
learning rate	0,0003
time horizon	128
beta	0,005
buffer size	2048
epsilon	0,2
gamma	0,99
lambd	0,95
max steps	2M
normalize	false
use recurrent	false
use curiosity	false
num epoch	3

Na Seção 4.3 apresentamos os cinco tipos de representação do estado observado. Vimos que, dependendo do tipo de representação, o tamanho do vetor de observação aumenta ou diminui. Como a grade dos cenários do BLE tem 9 linhas por 9 colunas e há mais duas observações quanto a posição (x, y) normalizada do agente, podemos concluir que o tamanho do vetor de observação para cada tipo de representação será:

1. Flag Binária = $81 + 2 = 83$ observações (tamanho da grade + posição)
2. Flag Binária Normalizada = $81 + 2 = 83$ observações (tamanho da grade + posição)
3. Híbrida = $7 * 81 + 2 = 569$ observações (tamanho do vetor híbrido $\times$ tamanho da grade + posição)
4. ICAART = $4 * 81 + 2 = 326$ observações ($4 \times$ tamanho da grade + posição)
5. ZeroOuUm = $7 * 81 + 2 = 569$ observações ($7 \times$ tamanho da grade + posição)

Para cada um dos cinco treinamentos de cada tipo de representação foram geradas as médias e o desvio padrão para cada estatística mencionada anteriormente: *recompensa acumulada, entropia, tamanho do episódio, loss de política, loss de valor e estimativa de valor*. Todos os gráficos contendo as estatísticas do treinamento foram suavizados em média com um passo de 100K por registro. Nesta seção, analisamos se o treinamento se comportou da maneira esperada quanto a estatística de recompensa acumulada, pois utilizamos esta para escolher os modelos que seriam usados pelos Brains durante a execução dos testes (torneios). A análise das demais estatísticas podem ser encontradas no Apêndice C na Seção C.1.

Ao analisarmos o gráfico da Figura 5.4, notamos que a recompensa acumulada, a partir da iteração 10K a 601K, dos Brains de Flag Binária e Flag Binária Normalizada ficam isolados dos outros Brains por quase todo o início do treinamento, alcançando o desempenhos médio dos outros ao final do treinamento. Enquanto isso, os outros Brains ficam revezando a dianteira até a iteração 651K, onde o Brain ICAART obtém a melhor recompensa acumulada média e ultrapassa os demais até o final do treinamento. A recompensa acumulada média tanto dos Brains ZeroOuUm e Híbrido foram semelhantes. Note que o desvio padrão se comporta de forma similar para todos os Brains.

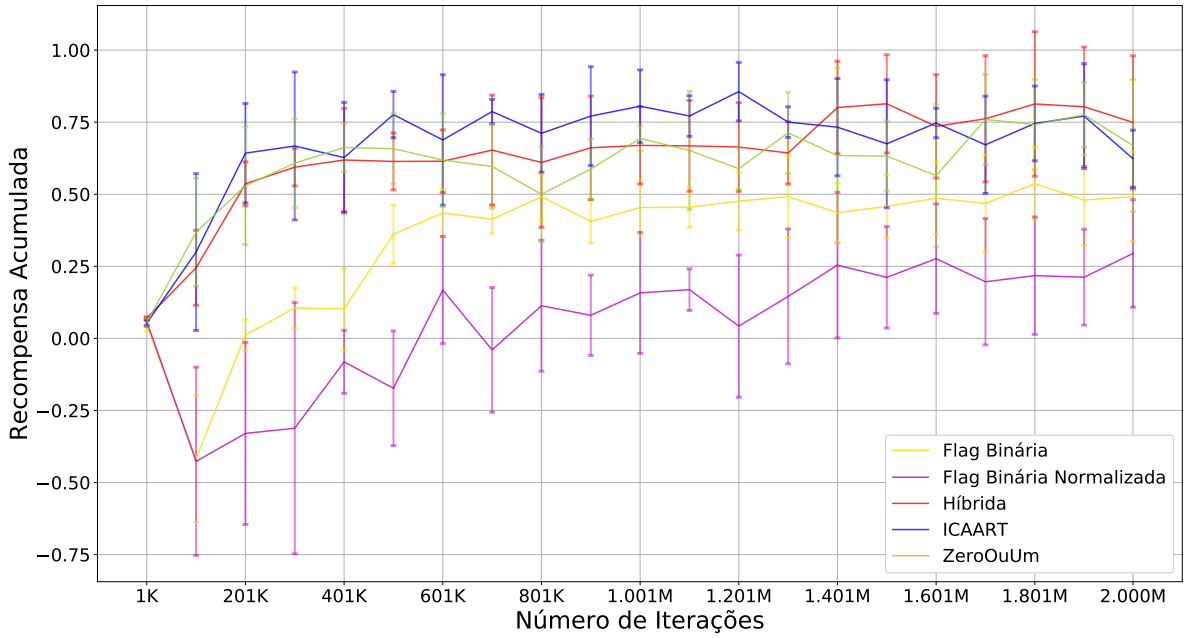

Figura 5.4: Gráfico da Recompensa Acumulada obtido nos treinamentos no ambiente Bomberman comparando os diferentes tipos de representação do estado. Desvio padrão é mostrado por linhas verticais.

Após a conclusão dos treinamentos, para se comparar qual agente/Brain é o mais vitorioso, foram criados alguns torneios (batalhas entre os agentes) e os resultados destes torneios são apresentados por meio de tabelas que trazem informações tais como: número e porcentagem de empates, e número e porcentagem de vitória de cada Brain em cada cenário e/ou no total. O nome do Brain é informado como sendo: letras iniciais do nome da representação concatenado com o número do modelo gerado na fase de treinamento. Os cenários são enumerados de acordo com o que foi explicado na Seção 5.1.2. T1000 são torneios com 1000 partidas para cada cenário do BLE. T100 são torneios com 100 partidas para cada cenário do BLE.

Após os treinamentos, implementamos um torneio com o objetivo de descobrir qual é a representação de estado que gera Brains mais vitoriosos. Por conta disso, criamos o **T1000 Vencedores** (ver Tabela 5.11). A Figura 5.4 mostra a recompensa acumulada média para cada elemento em *homogêneo_brain* para suas cinco execuções de treinamento. Ao inspecionar as recompensas acumuladas obtidas no final de cada execução de treinamento, descartamos aquele com o pior valor, ou seja, selecionamos o elemento $FB_i \in FB$ com a menor recompensa acumulada, o elemento $FBN_j \in FBN$ com a menor recompensa acumulada e assim por diante. Estes são descartados de seus respectivos conjuntos, gerando novos conjuntos com quatro elementos, por exemplo, $FB^* = FB - FB_i$, $FBN^* = FBN - FBN_j$. Os agentes dentro de cada um desses novos conjuntos S^* vão lutar contra os outros agentes em seu mesmo conjunto em um torneio de 100 partidas, chamado $T100$. Os torneios são realizados com quatro agentes, porque esse é o número máximo de agentes permitidos em um mesmo cenário do BLE. Durante o desenvolvimento desta dissertação, fizemos um experimento prévio com o objetivo de descobrirmos o melhor tipo de representação de estado, porém, escolhemos os participantes do torneio T1000 deste experimento prévio como sendo o representante com maior recompensa acumulada de cada tipo de representação. Julgamos essa abordagem ingênua porque com ela testamos apenas um modelo de cada representação num torneio final, sem ao menos testarmos qual modelo entre os modelos de um tipo de representação é de fato o melhor. Os resultados desse experimento prévio podem ser encontrados no apêndice C na Seção C.1.

O resultado final do **T100 de Flag Binária** pode ser visto na Tabela 5.5; do **T100 Flag Binária Normalizada** na Tabela 5.6, do **T100 Híbrido** na Tabela 5.7, do **T100 ICAART** na Tabela 5.8, e do **T100 ZeroOuUm** na Tabela 5.9. Novamente, como temos no máximo quatro agentes em um cenário, devemos descartar o pior desses 5 para compor o conjunto dos modelos que batalharão no torneio **T1000 Vencedores**. No entanto, como os Brains de Flag Binária e de Flag Binária Normalizada obtiveram uma

recompensa acumulada muito próxima uma da outra ao término dos seus treinamentos, fizemos um T100 entre eles (**T100 Flag Binária Versus Flag Binária Normalizada**) em que 2 dos 4 agentes em cada cenário usam o Brain FB e os outros 2 dos 4 usam o Brain FBN. Nesse torneio, o modelo utilizado pelo FB foi o modelo vencedor do **T100 Binary Flag**, e o modelo utilizado pelo FBN foi o modelo vencedor do **T100 Binary Flag Normalizada**. Se qualquer um dos agentes do FB vence então a vitória é do FB, caso qualquer um dos agentes FBN vence, a vitória é do FBN. Os resultado desse torneio está na Tabela 5.10.

Na Tabela 5.5 observa-se que o segundo modelo do treinamento do Brain de Flag Binária obteve 285 vitórias e o quinto modelo que possui a melhor recompensa acumulada ficou em último lugar. Os modelos 4 e 3 estão tecnicamente empatados em segundo lugar (228 e 233 vitórias respectivamente). Além disso, uma informação que pode ser acrescentada, é que nenhum agente que usou o Brain de Flag Binária na fase de inferência "soube" jogar o jogo Bomberman corretamente. Os agentes se suicidavam em poucas iterações ou ficavam andando contra a parede ou contra os blocos.

Tabela 5.5: Resultado do T100 Flag Binária. Total e porcentagem de Vitórias considerando todos cenários. Os participantes deste torneio foram os Brains com os modelos 5, 2, 4 e 3 obtidos na fase de treinamento.

Resultado	**Número de Vitórias**
Empate	78 (7,8%)
Modelo 5	176 (17,6%)
Modelo 2	**285 (28,5%)**
Modelo 4	228 (22,8%)
Modelo 3	233 (23,3%)
Total	1000 (100%)

Os resultados do Brain de Flag Binária Normalizada podem ser observados na Tabela 5.6, sendo que os modelos 1 e 2 obtiveram resultados semelhantes, com 282 e 256 vitórias respectivamente. Contudo, o modelo vencedor foi o primeiro. Também com esse tipo de representação, os agentes não "souberam" jogar o jogo Bomberman, embora alguns até conseguiram destruir alguns blocos e fugir de algumas bombas.

Tabela 5.6: Resultado do T100 Flag Binária Normalizada. Total e porcentagem de Vitórias considerando todos cenários. Os participantes deste torneio foram os Brains com os modelos 4, 1, 5 e 2 obtidos na fase de treinamento.

Resultado	**Número de Vitórias**
Empate	94 (9,4%)
Modelo 4	172 (17,2%)
Modelo 1	**282 (28,2%)**
Modelo 5	196 (19,6%)
Modelo 2	256 (25,6%)
Total	1000 (100%)

Na Tabela 5.7, são apresentados os resultados do T100 entre os modelos com representação híbrida. Os modelos 1 e 3 obtiveram os melhores resultados, com 385 e 343 vitórias respectivamente. O modelo vitorioso foi o primeiro modelo. Os agentes que usam esse tipo Brain aprenderam a colocar e fugir de bombas.

Tabela 5.7: Resultado do T100 Híbrido. Total e porcentagem de Vitórias considerando todos cenários. Os participantes deste torneio foram os Brains com os modelos 2, 3, 1 e 4 obtidos na fase de treinamento.

Resultado	**Número de Vitórias**
Empate	41 (4,1%)
Modelo 2	120 (12%)
Modelo 3	343 (34,3%)
Modelo 1	**385 (38,5%)**
Modelo 4	111 (11,1%)
Total	1000 (100%)

O quarto modelo treinado com Brain ICAART foi o modelo que obteve o maior número de vitórias entre seus similares, além de ter sido o modelo que obteve a maior recompensa acumulada no final de sua fase de treinamento. Foram obtidas 272 vitórias (ver Tabela 5.8). Este Brain conseguiu aprender a usar bombas e a fugir delas efetivamente. Note que o segundo modelo obteve um número de vitórias próximo ao do primeiro colocado (266 vitórias). Observamos que este torneio foi disputado, ao se analisar os resultados.

Tabela 5.8: Resultado do T100 ICAART. Total e porcentagem de Vitórias considerando todos cenários. Os participantes deste torneio foram os Brains com os modelos 4, 3, 1 e 2 obtidos na fase de treinamento.

Resultado	**Número de Vitórias**
Empate	37 (2,3%)
Modelo 4	**272 (27,2%)**
Modelo 3	239 (23,9%)
Modelo 1	186 (18,6%)
Modelo 2	266 (26,6%)
Total	1000 (100%)

Na Tabela 5.9, foi observado que o primeiro modelo treinado com o Brain ZeroOuUm obteve o maior número de vitórias (297) e em segundo lugar ficou o terceiro modelo treinado (275). A disputa pelo primeiro lugar foi acirrada. Os agentes que utilizam esse Brain também aprenderam a usar bombas e a fugir delas efetivamente.

Tabela 5.9: Resultado do T100 ZeroOuUm. Total e porcentagem de Vitórias considerando todos cenários. Os participantes deste torneio foram os Brains com os modelos 1, 4, 3 e 5 obtidos na fase de treinamento.

Resultado	**Número de Vitórias**
Empate	47 (4,7%)
Modelo 1	**297 (29,7%)**
Modelo 4	172 (17,2%)
Modelo 3	275 (27,5%)
Modelo 5	209 (20,9%)
Total	1000 (100%)

Como explicado anteriormente, para não existir injustiça com o Brain de Flag Binária, foi realizado um T100 entre Flag Binária e Flag Binária Normalizada (ver Tabela 5.10) para se saber quem ganharia a última vaga do **T1000 Vencedores**. Contrariando o resultado obtido no torneio prévio entre os dois tipos de representação mostrado na Tabela C.1, o Brain de Flag Binária Normalizada saiu vitorioso com $63,5\%$ das vitórias e garantiu sua vaga no **T1000 Vencedores**. Note que $9,4\%$ dos resultados foram empate. Essa diferença entre os dois Brains pode ser explicada porque quando normalizamos as

flags binárias, seus valores ficam entre o intervalo [0, 1], o que geralmente é recomendado para algoritmos que utilizam redes neurais [53]. Os valores do Brain de Flag Binária não são normalizados e podem chegar a valores altos, como 128 por exemplo. Os agentes treinados com representação de FBN, mesmo utilizando este tipo de representação que normaliza as observações que são enviadas para o vetor de observações, não aprenderam a jogar o Bomberman corretamente (e.g. colocar bomba e fugir de bomba), porém se saíram melhor que os agentes que utilizaram a representação de FB. Talvez, estes agentes de FBN não aprendem a jogar o Bomberman porque a representação FBN não segue a outra recomendação do ML-Agents, que diz que para observações de tipos enumerados de dados, a representação One-Hot deve ser utilizada[53], e o tipo FBN não é nem um pouco similar a representação One-Hot.

Tabela 5.10: Resultado do T100 Flag Binária Versus Flag Binária Normalizada. Total e porcentagem de Vitórias considerando todos cenários. Os participantes deste torneio foram os Brains Vencedores Flag Binária 2 e de Flag Binária Normalizada 1. Note que para cada Brain haviam dois agentes

Resultado	**Número de Vitórias**
Empate	94 (9,4%)
FB_2	271 (29,8%)
$\boldsymbol{FBN_1}$	**635 (63,5%)**
Total	1000 (100%)

No último torneio deste experimento, para se saber qual representação dos estados é a melhor para o ambiente do Bomberman, foi realizado um **T1000 Vencedores** (ver Tabela 5.11) em que seus participantes foram os Brains vencedores dos **T100** das Tabelas 5.7, 5.8, 5.9 e 5.10. O primeiro modelo treinado com o Brain de representação Híbrida (H_1) foi o Brain que obteve o maior número de vitórias (4442), cerca de $44,42\%$ dos resultados.

Tabela 5.11: Resultado do T1000 Vencedores. Total e porcentagem de Vitórias considerando todos cenários. Os participantes deste torneio foram os Brains: FBN_1, H_1, I_4 *e* Z_1

Resultado	**Número de Vitórias**
Empate	263 (2,63%)
FBN_1	319 (3,19%)
$\boldsymbol{H_1}$	**4442 (44,42%)**
I_4	1782 (17,82%)
Z_1	3194 (31,94%)
Total	10000 (100%)

Portanto, foi concluído que dentre as formas analisadas, a melhor forma de se representar o espaço de estados de uma grade de um jogo estilo Bomberman, para um agente que utilize um algoritmo de DRL como o PPO, é a forma de representação híbrida da grade, onde os valores sempre estão na faixa entre 0 e 1. A segunda melhor forma é a representação ZeroOuUm e a terceira a do ICAART.

5.3 Treinamento Multi-Brain

Em resumo, o que fizemos na primeira parte deste experimento é mostrado na Figura 5.5. Fizemos 5 treinamentos Multi-Brain, escolhemos os 4 modelos de cada tipo de representação com maior recompensa acumulada, realizamos T100s por representação para descobrirmos os participantes do **T1000 Vencedores Multi-Brain**, e por fim o vencedor deste torneio foi o $M.H_4$.

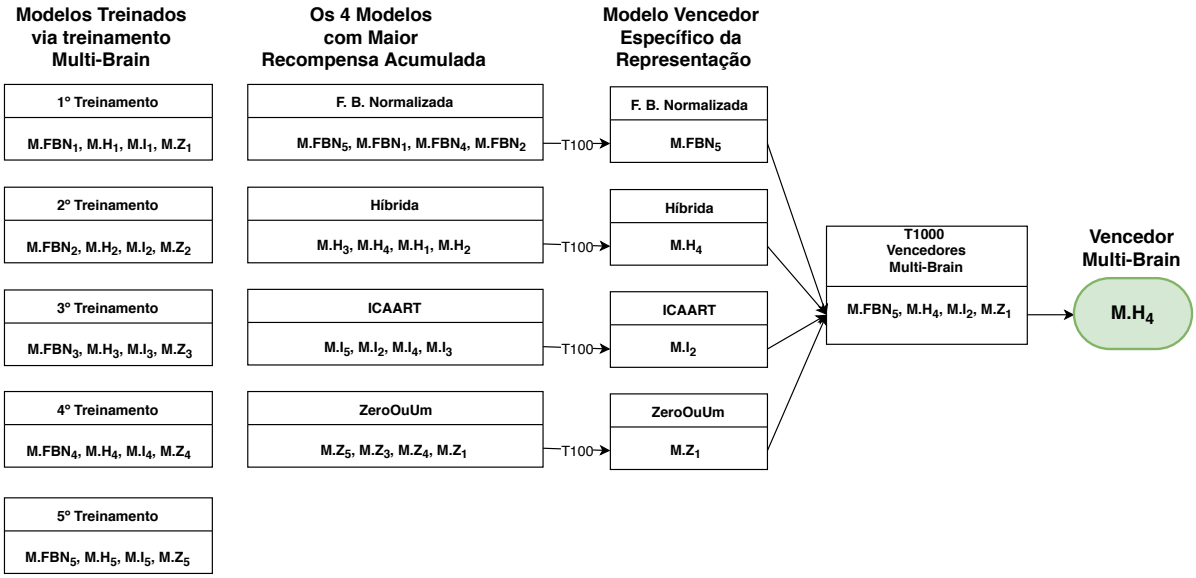

Figura 5.5: Fluxo do Experimento Multi-Brain até o resultado do T1000 Vencedores Multi-Brain.

Nesta subseção apresentamos o experimento do treinamento Multi-Brain e seus resultados. Num treinamento Multi-Brain, um ou mais Brains são treinados ao mesmo tempo no mesmo ambiente. Com isso, podemos, com um treinamento só, gerar vários modelos de uma vez, um para cada Brain que participou do treinamento (quantidade de Brains) ao invés de apenas um modelo, como no treinamento Um-Brain. Além disso, no treinamento Multi-Brain, um Brain já treina com oponentes diferentes dele.

Nessa pesquisa foram consideradas algumas possibilidades quanto à melhor forma de se fazer o treinamento dos agentes. Verificamos se era melhor treinar apenas um Brain com vários agentes atrelados a ele por vez (treinamento Um-Brain) ou se era melhor treinar quatro Brains de uma vez só (treinamento Multi-Brain), onde cada Brain seria responsável por um tipo de agente (para cada representação) em cada um dos dez cenários do BLE. Para descobrir qual é a melhor forma de treinamento entre as duas, foi realizado este outro experimento em que cada Brain foi responsável por um tipo de agente de cada um dos dez cenários. Note que não foi utilizado um Brain de Flag Binária nesse experimento porque esta representação de estado teve os piores resultados no experimento anterior. Os quatro Brains escolhidos para esse experimento foram, então: Flag Binária Normalizada, Híbrido, ICAART e ZeroOuUm. Neste caso, teremos um torneio entre 4 Brains com representações de estados diferentes e aprendendo diferentes funções de política. Assim, temos um conjunto *multi-brain = {MFBN, MH, MI, MZ}* correspondente ao tipo de representação escolhida pelo agente, que é repetido cinco vezes no total, produzindo cinco

modelos diferentes.

Os hiper-parâmetros usados neste experimento foram os mesmos do experimento anterior (quanto a melhor representação do estado). Os gráficos contendo as estatísticas desse treinamento podem ser visualizados nos Gráficos das Figuras (5.6 C.8 C.9 C.10 C.11). Todos os gráficos e análises das estatísticas, exceto das de recompensa acumulada, estão no Apêndice C e Seção C.2. Como há 4 Brains que batalham entre si durante o treinamento, configuramos tanto os treinamentos quanto os torneios para gerar de modo fixo 4 agentes por cenário.

No gráfico da Figura 5.6 foi observado que claramente uma política se sobrepõe as outras. Os Brains ZeroOuUm e Híbrido obtém ótimos resultados de recompensa acumulada, que extrapolam 1.0, enquanto que o ICAART e o de Flag Binária Normalizada não tiveram tendência de crescimento considerável. Note que, os resultados obtidos pelos dois Brains que obtiveram as melhores recompensas acumuladas nesse treinamento são bem maiores que os obtidos no gráfico da Figura 5.4, enquanto que os resultados obtidos pelos dois Brains que obtiveram as piores recompensas acumuladas são menores que os gráficos da Figura 5.4. Isso acontece porque, durante o treinamento, cada partida sempre começa com 4 agentes, fornecendo mais inimigos para serem mortos por um Brain individual na configuração de múltiplos Brains, permitindo assim mais recompensa. Outro dado que pode ser extraído deste gráfico, é que, praticamente, não há empate entre os Brains, exceto numa pequena faixa de pontos entre a representação Híbrida e a ZeroOuUm, quando se considera o desvio padrão.

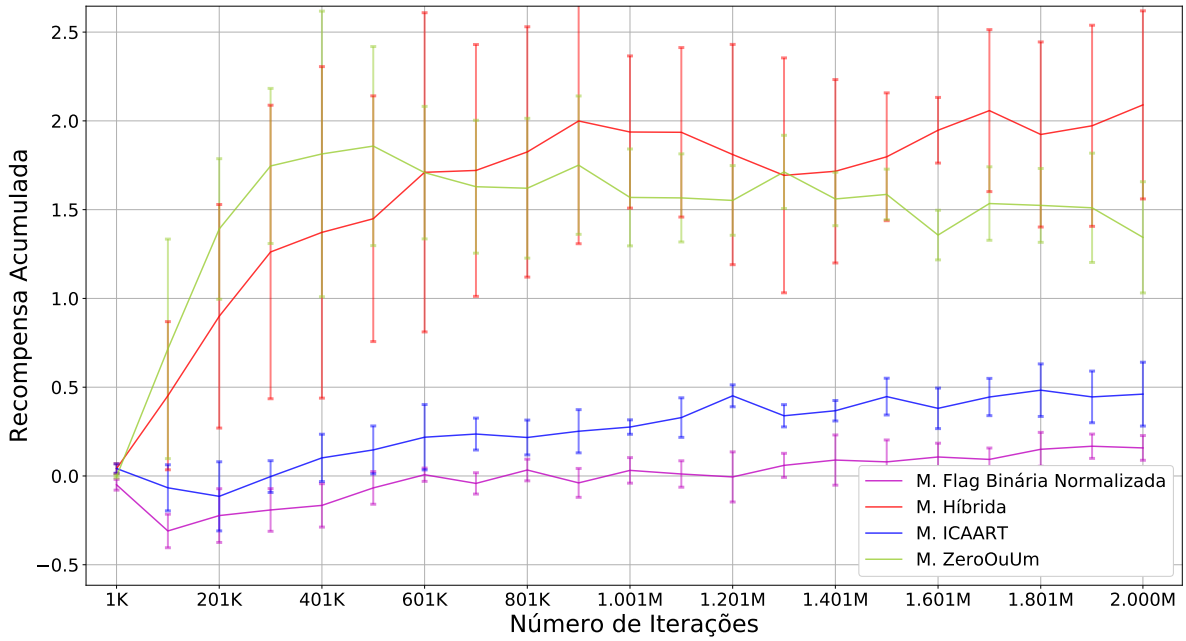

Figura 5.6: Gráfico da Recompensa Acumulada obtido nos treinamentos Multibrain no ambiente Bomberman.

Após o término do treinamento Multi-Brain, foram realizados torneios (T100 Multi-Brain) para cada tipo de representação, com os modelos que obtiveram as melhores recompensas acumuladas. Note que foram utilizados quatro dos cinco modelos gerados, porque há somente 4 vagas disponíveis por torneio no BLE, por isso escolhemos os 4 modelos que obtiveram as melhores recompensas acumuladas. Os resultados desses torneios podem ser encontrados nas Tabelas (5.12, 5.13, 5.14, 5.15). Conhecendo os vencedores destes torneios, selecionamos, para cada representação, os modelos que obtiveram mais vitórias, e com eles formamos o conjunto $multibrain_{best} = \{ M.NBF_{best}, M.H_{best}, M.I_{best}, M.Z_{best}\}$ que será usado no torneio **T1000 Vencedores Multi-Brain**. Este conjunto foi instanciado como $multibrain_{best} = \{ M.NBF_5, M.H_4, M.I_2, M.Z_1\}$. Os resultados deste torneio são exibidos na Tabela 5.16. Por fim, os dois primeiros colocados do **T1000 Vencedores Multi-Brain** foram escolhidos para batalhar contra os dois primeiros colocados do **T1000 Vencedores** (ver Tabela 5.11) em um novo torneio, que é o **T1000 One-Brain vs Multi-Brain** (Tabela 5.17), com o objetivo de comparar a melhor forma de se treinar os agentes: individualmente ou utilizando múltiplos Brains.

No primeiro Torneio 100, do Brain ZeroOuUm, mostrado na Tabela 5.12, o primeiro modelo Multi-Brain que foi gerado nos treinamentos, obteve a maior porcentagem de vitórias ($33,3\%$) ficando na frente do segundo modelo que obteve $31,6\%$ das vitórias e do terceiro que obteve $30,6\%$. Pode-se considerar que houve um empate entre os três

primeiros, porém, o modelo que se classificou para o **Torneio 1000 Vencedores Multi-Brain** foi o terceiro modelo. Note que o desempenho do quarto modelo treinado foi inferior aos demais e a porcentagem de empates foi muito pequena.

Tabela 5.12: Resultado do T100 Multi-brain ZeroOuUm. Total e porcentagem de Vitórias considerando todos cenários. Os participantes deste torneio foram os Brains com os modelos 5, 3, 4 e 1 obtidos na fase de treinamento (ordem decrescente considerando recompensa acumulada).

Resultado	**Número de Vitórias**
Empate	8 (0.8%)
Modelo 5	316 (31,6%)
Modelo 3	306 (30,6%)
Modelo 4	37 (3,7%)
Modelo 1	**333 (33,3%)**
Total	1000 (100%)

No segundo T100 realizado, quanto ao Brain Híbrido, cujos resultados são mostrados na Tabela 5.13, o quarto modelo gerado nos treinamentos obteve 56% das vitórias, ficando a frente do segundo colocado que obteve 19%. O resultado deste torneio é peculiar porque o primeiro colocado destoou bastante dos outros oponentes.

Tabela 5.13: Resultado do T100 Multi-Brain Híbrido. Total e porcentagem de Vitórias considerando todos cenários. Os participantes deste torneio foram os Brains com os modelos 3, 4, 1 e 2 obtidos na fase de treinamento (ordem decrescente considerando recompensa acumulada).

Resultado	**Número de Vitórias**
Empate	32 (3,2%)
Modelo 3	190 (19%)
Modelo 4	**560 (56%)**
Modelo 1	92 (9,2%)
Modelo 2	126 (12,6%)
Total	1000 (100%)

O vitorioso do terceiro T100 que foi realizado para se descobrir qual modelo será

usado pelo Brain ICAART foi o segundo modelo gerado nos treinamentos, obtendo 30% das vitórias. Uma observação que pode ser adicionada ao resultado desse torneio, é que, na prática, os agentes não conseguiram aprender a usar bombas corretamente. Eles ficavam presos nas células próximas às suas posições inciais. Por isso, a porcentagem de empates foi alta (13, 6%).

Tabela 5.14: Resultado do T100 Multi-Brain ICAART. Total e porcentagem de Vitórias considerando todos cenários. Os participantes deste torneio foram os Brains com os modelos 5, 2, 4 e 3 obtidos na fase de treinamento (ordem decrescente considerando recompensa acumulada).

Resultado	**Número de Vitórias**
Empate	136 (13,6%)
Modelo 5	154 (15,4%)
Modelo 2	**300 (30%)**
Modelo 4	229 (22,9%)
Modelo 3	181 (18,1%)
Total	1000 (100%)

No último T100 realizado, para se descobrir qual o modelo que será usado pelo Brain de Flag Binária Normalizada, cujos resultados são mostrados na Tabela 5.15, o modelo vitorioso foi o quinto modelo gerado na fase de treinamento. Ele obteve 30, 8% das vitórias. Note que nesse torneio, na prática, os agentes não aprenderam a usar bombas efetivamente. Quando colocavam bombas, não se protegiam delas, e as vezes conseguiam fugir de algumas.

Tabela 5.15: Resultado do Torneio 100 Multi-Brain Flag Binária Normalizada. Total e porcentagem de Vitórias considerando todos cenários. Os participantes deste torneio foram os Brains com os modelos 5, 1, 4 e 2 obtidos na fase de treinamento (ordem decrescente considerando recompensa acumulada).

Resultado	**Número de Vitórias**
Empate	131 (13,1%)
Modelo 5	**308 (30,8%)**
Modelo 1	170 (17%)
Modelo 4	154 (15,4%)
Modelo 2	237 (23,7%)
Total	1000 (100%)

Com os quatro participantes já definidos, o **Torneio 1000 Vencedores Multi-Brain** foi realizado. Seus resultados estão na Tabela 5.16. Como se pode observar, o Brain vitorioso foi o $M.H_4$ que obteve $54,22\%$ das vitórias. Em segundo lugar, ficou o Brain $M.Z_1$ que obteve $23,47\%$. Esses dois Brains se classificaram para o **T1000 Vencedores vs Vencedores Multi-Brain**. Note que o desempenho do $M.I_2$ foi bem inferior ao desempenho obtido por ele quando o treinamento foi individual. Observou-se que ele não aprendeu a usar bombas e nem a fugir delas neste torneio, por isso seu resultado foi abaixo do esperado. Também, pode ser observado que o Brain $M.FBN_5$ ficou em terceiro lugar, a frente até mesmo do ICAART. Isso se deve ao fato de que este Brain aprendeu a fugir de bombas em algumas configurações dos estados das células do grid, tornando-se um pouco competitivo, porque desta forma ele consegue, muitas vezes, ser o último agente vivo, evitando ser abatido por bombas. O que precisa ficar evidente neste treinamento Multi-Brain é que, na prática, o desempenho tanto do ICAART quanto Brain de Flag Binária Normalizada ficou bem abaixo do esperado.

Tabela 5.16: Resultado do T1000 Vencedores Multi-Brain. Total e porcentagem de Vitórias considerando todos cenários. Os participantes deste torneio foram os Brains: $M.Z_1$, $M.H_4$, $M.FBN_5$ *e* $M.I_2$ respectivamente. Note que a letra M que vem antes do nome do Brain diz respeito aos modelos multi-brain.

Resultado	**Número de Vitórias**
Empate	578 (5,78%)
$M.Z_1$	2347 (23,47%)
$\boldsymbol{M.H_4}$	**5422 (54,22%)**
$M.FBN_5$	1313 (13,13%)
$M.I_2$	340 (3,4%)
Total	10000 (100%)

Para se comparar qual é a melhor forma de se treinar os agentes, se individualmente ou utilizando Multi-Brains, foi realizado um último torneio que foi intitulado de **T1000 Vencedores vs Vencedores Multi-Brain**. Os resultados desse torneio podem ser visualizados na Tabela 5.17. Os participantes desse torneio foram os dois primeiros colocados do **T1000 Vencedores** e os dois primeiros colocados do **T1000 Vencedores Multi-Brain**. Pode ser observado nos resultados deste torneio que o desempenho dos Brains treinados individualmente foram superiores aos dos Brains que foram treinados com outros Brains ao mesmo tempo. O modelo vitorioso deste torneio foi novamente o Brain H_1 e o segundo colocado foi novamente o Brain Z_1. Eles tiveram $30,74\%$ e $27,79\%$ das vitórias respectivamente.

Tabela 5.17: Resultado do T1000 Vencedores vs Vencedores Multi-Brain. Total e porcentagem de Vitórias considerando todos cenários. Os participantes deste torneio foram os dois primeiros colocados do T1000 Vencedores: H_1 e Z_1, e os dois primeiros colocados do T1000 Vencedores Multi-brain: $M.H_4$ e $M.Z_1$. Note que a letra M antes do nome do Brain denota que é Multi-Brain.

Resultado	**Número de Vitórias**
Empate	544 (5,44%)
H 1	**3074 (30,74%)**
Z 1	2779 (27,79%)
M.H 4	2472 (24,72%)
M.Z 1	1131 (11,31%)
Total	10000 (100%)

Resumidamente, o fluxo do torneio entre os 2 primeiros colocados do **T1000 Vencedores** e os 2 primeiros colocados do **T1000 Vencedores Multi-Brain** é mostrado na Figura 5.7. O vencedor foi o H_1.

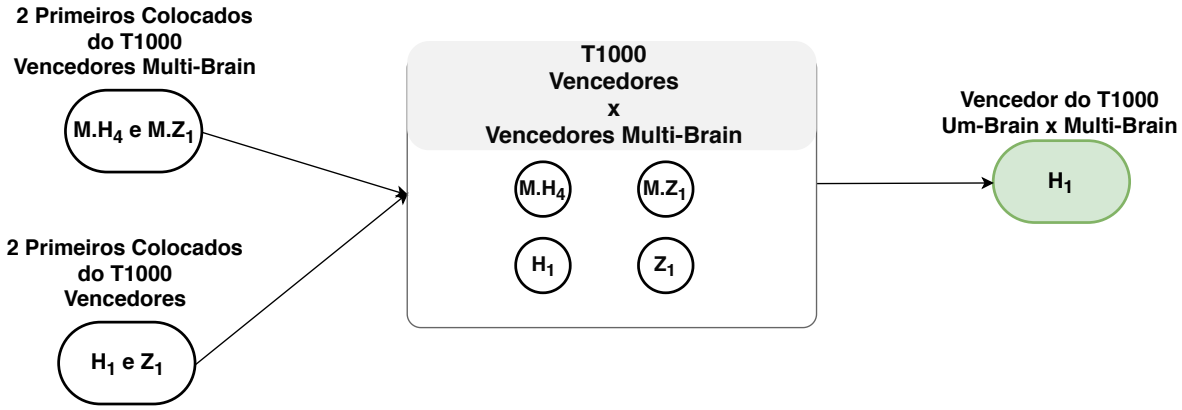

Figura 5.7: Fluxo da Segunda Parte do Experimento Multi-Brain em que colocamos para batalhar num mesmo torneio os 2 primeiros colocados do T1000 Vencedores e os 2 primeiros colocados do T1000 Vencedores Multi-Brain.

Portanto, foi concluído que no BLE, treinar cada Brain individualmente (Um-Brain) é melhor do que treinar vários Brains ao mesmo tempo (Multi-Brain). Essa conclusão pode ser estendida para outros ambientes e cenários, uma vez que os dados obtidos indicam que quando se faz um treinamento Multi-Brain, uma política irá se sobrepor às outras, gerando bons resultados estatísticos na fase de treinamento, superiores às estatísticas geradas pelo

treinamento Um-Brain. Porém, quando se colocam agentes treinados com ambas técnicas para batalharem entre si, foi observado que os agentes Um-Brain obtiveram os melhores resultados. Além disso, treinamento Um-Brain explora mais o espaço de estados, porque neste treinamento, para cada um dos dez cenários, há até quatro agentes aprendendo a jogar, enquanto que no treinamento Multi-Brain cada um é responsável por apenas um agente por cenário. Em resumo, num treinamento Um-Brain, há mais exemplos para alimentar a rede, onde a experiência de todos os agentes de cada cenário é somada ao conjunto de exemplos, fazendo com que a aprendizagem seja melhor. Poderíamos ter dobrado o tempo de treinamento para compensar a falta de experiência coletada dos modelos treinados via treinamento Multi-Brain em comparação com os do Um-Brain, contudo, como dito anteriormente, a política de algum Brain acabaria se sobrepondo as outras, e esse Brain que possui esta política vitoriosa não necessariamente seria um Brain que fosse capaz de derrotar um outro que tivesse treinado via treinamento Um-Brain.

5.4 Treinamento Um-Brain com PPO e LSTM

De forma resumida, o fluxo de todo este experimento é demonstrado na Figura 5.8. Primeiramente, treinamos os modelos com LSTM via treinamento Um-Brain, escolhemos os 4 modelos com maior recompensa acumulada, realizamos um T100 para descobrir qual modelo é mais vitorioso, e por fim fazemos um **T1000 LSTM x Híbrida** para descobrir se o modelo vitorioso entre os LSTMs vencerá o campeão do torneio **T1000 Vencedores**.

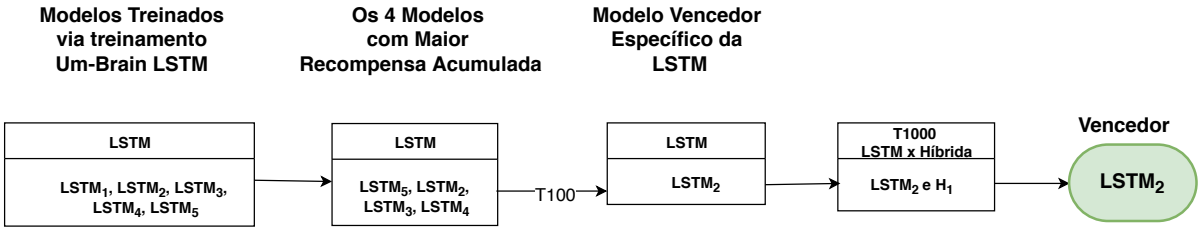

Figura 5.8: Fluxo do experimento do LSTM

Nesta subseção foram realizados experimentos com Brains que utilizam a representação Híbrida e treinam com o método PPO com LSTM ativada para verificar se as redes recorrentes influenciam positivamente a aprendizagem de agentes que utilizam o PPO. Escolhemos a representação Híbrida porque os Brains vencedores dos T1000 realizados anteriormente usaram esse tipo de representação. Além disso, utilizamos o treinamento Um-Brain porque por meio de experimentos confirmamos que este tipo de treinamento produz agentes mais vitoriosos. No treinamento de agentes com ML-Agents, ao se ativar

a LSTM para ser usada como aproximadora de função do algoritmo PPO, o treinamento fica mais pesado, pois a quantidade de cálculos aumenta consideravelmente. Por isso, foi necessário diminuir alguns hiper-parâmetros do PPO para que fosse possível executar o treinamento. Veja a Tabela 5.18 contendo os hiper-parâmetros alterados.

Tabela 5.18: Hiperparâmetros do PPO: LSTM

Nome	**Valor**
num layers	1
time horizon	64
buffer size	1024
use recurrent	true
sequence length	32
memory size	256

Neste experimento, o treinamento Um-Brain com o método PPO e LSTM e utilizando a representação híbrida foi abreviado resumidamente como LSTM. Os gráficos contendo as estatísticas dos treinamentos deste experimento podem ser encontrados na Seção C.3 do Apêndice C. Em resumo, todas as estatísticas deste treinamento foram semelhantes às estatísticas do primeiro experimento utilizando a representação Híbrida.

Após os treinamentos, realizamos um novo torneio, **T100 LSTM** (Tabela 5.19), para descobrirmos qual modelo LSTM seria o vencedor dentre os cinco modelos gerados durante os treinamentos. O mesmo critério anterior foi utilizado, ou seja, os quatro modelos com melhor recompensa acumulada participaram deste torneio. O modelo vencedor foi o segundo modelo treinado com o Brain LSTM, que obteve $31,4\%$ das vitórias.

Tabela 5.19: Resultado do T100 LSTM. Os participantes deste torneio foram os Brains LSTM com os modelos 5, 2, 3 e 4 obtidos na fase de treinamento (ordem decrescente considerando recompensa acumulada).

Resultado	**Número de Vitórias**
Empate	38 (3,8%)
Modelo 5	142 (14,2%)
Modelo 2	**314 (31,4%)**
Modelo 3	249 (24,9%)
Modelo 4	257 (25,7%)
Total	1000 (100%)

Por fim, realizamos um novo torneio, **T1000 LSTM vs Híbrida**, para descobrirmos se ativando o LSTM no algoritmo PPO melhoraria o desempenho de um agente que utiliza a representação Híbrida para observar o estado da grade do jogo. Cada Brain foi responsável por dois agentes em cada cenário. O Brain Híbrido utilizado foi o vencedor do **T1000 One-Brain vs Multi-Brain**. Como pode ser visto na Tabela 5.20, o segundo modelo treinado com o Brain LSTM venceu o torneio com $61,02\%$ das vitórias.

Tabela 5.20: Resultado do T1000 LSTM vs Híbrido. Os participantes deste torneio foram os Brains $LSTM_2$ e H_1. Note que para cada Brain haviam dois agentes

Resultado	**Número de Vitórias**
Empate	395 (3,95%)
$\boldsymbol{LSTM_2}$	**6102 (61,02%)**
H_1	3503 (35,03%)
Total	10000 (100%)

Portanto, concluímos que ativar LSTM no algoritmo PPO implementado pelo ML-Agents produz agentes/Brains mais vitoriosos. Isso se deve à capacidade de memória das LSTMs, que permitem que o agente aprenda o que fazer, mesmo quando a recompensa está num futuro distante. O resultado obtido neste torneio foi considerável, pois o primeiro modelo de representação Híbrida havia vencido todos os outros modelos em todos os outros torneios anteriores.

5.5 Método com BC e PPO

O fluxograma contendo, de forma resumida, todo experimento do BC até esta parte é mostrado na Figura 5.9.

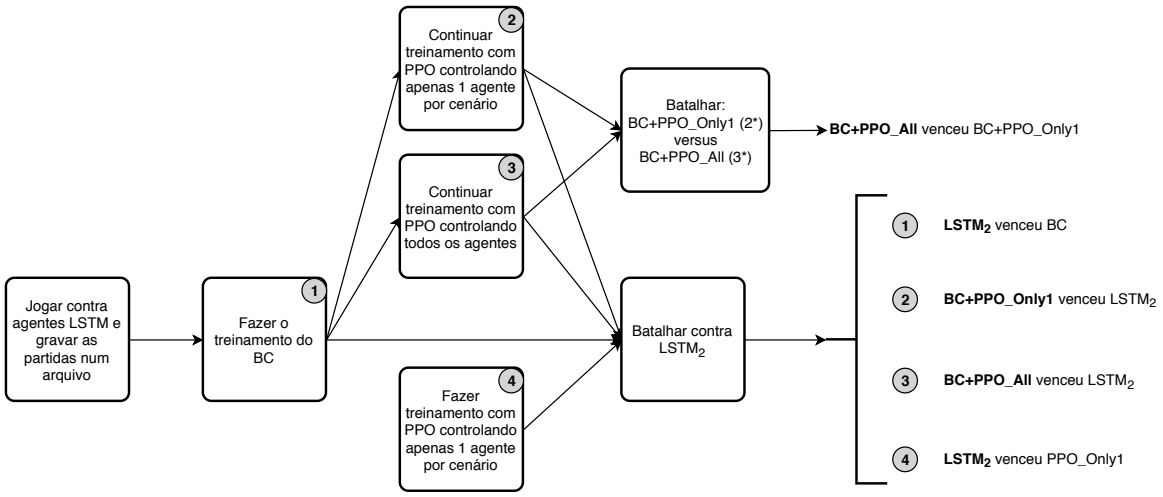

Figura 5.9: Fluxo do experimento do BC+PPO

Finalmente, treinamos um agente com o algoritmo BC com a representação Híbrida e continuamos seu treinamento com o PPO. Antes de iniciarmos o treinamento, gravamos 40 partidas em um arquivo de repetição onde jogamos[4] pelo menos 3 partidas em cada cenário estático e 20 partidas em cenários aleatórios, lutando contra outras entidades que usaram o Brain LSTM (vencedor do **T1000 LSTM vs Híbrido**). Estas partidas utilizadas no arquivo de repetição foram partidas em que o jogador venceu ou quase venceu (ficou vivo por pelo menos 20 iterações). Este arquivo de repetição age durante o treinamento do BC como o especialista, para que o agente aluno possa aprender a jogar com ele. No treinamento BC, usamos os mesmos hiper-parâmetros usados na Tabela 5.4, exceto que mudamos **max steps** para 100K porque o treinamento não precisa ser tão grande quanto um treinamento do PPO que tenta maximizar as recompensas obtidas, e adicionamos e configuramos **batches per epoch** para 5, que é um parâmetro específico do algoritmo BC que informa o número de lotes de exemplos de treinamento a serem coletados antes de treinar o modelo. O arquivo de repetição fornece o número de blocos destrutíveis e suas posições em cada cenário, o número de agentes e suas posições iniciais, as ações dos especialistas e inimigos em cada iteração, etc. Se o agente aluno morrer ou se o jogo durar mais de 400 iterações para acabar, redefinimos o cenário e carregamos a próxima partida do arquivo de repetição. Caso não haja mais partidas no arquivo de

[4]Apenas as partidas de um jogador foram utilizadas. O jogador foi o Ícaro Goulart

repetição, carregaremos a primeira partida do arquivo novamente. O especialista apenas envia ao algoritmo BC todas as suas demonstrações presentes em cada partida por 2 vezes no máximo.

Depois de terminar o treinamento com o BC, ainda podemos melhorar a função de política do agente de duas maneiras diferentes: **i) (BC + PPO_Only1):** executamos o PPO a partir da política aprendida pelo BC para treinar um agente contra inimigos que foram treinados com o modelo de um Brain PPO-LSTM e utilizando a representação de estado Híbrida. **ii) (BC + PPO_All):** No segundo modo de treinamento continuado, o PPO é executado após o BC, tal como foi feito antes, porém agora todos os agentes no cenário estão aprendendo dentro de um único Brain. Criamos uma série de torneios T100 e T1000 para observar o comportamento dos agentes BC e descobrimos o seguinte:

- **(T100-1vs1-BC-vs-LSTM):** O agente BC joga razoavelmente somente quando ele joga nos exatos cenários que ele treinou anteriormente, e quando seus inimigos se comportam da mesma maneira que está no arquivo de repetição. Quando o agente BC batalha contra o Brain LSTM em um torneio T100 1 versus 1, ele vence apenas 9.7% das partidas (Veja Tabela 5.21). Um T100 1 vs 1 é um T100 que possui apenas 2 agentes por cenário que são posicionados aleatoriamente no início da partida em um dos 4 cantos do cenário e estão atrelados à Brains diferentes.

Tabela 5.21: Resultado do T100 BC vs LSTM. Os participantes deste torneio foram os Brains $LSTM_2$ e BC. Note que para cada Brain existia apenas 1 agente

Resultado	**Número de Vitórias**
Empate	40 (4%)
$\boldsymbol{LSTM_2}$	**863 (86,3%)**
BC	97 (9,7%)
Total	1000 (100%)

- **(T100 1vs1 BC+PPO_Only1 vs LSTM):** Neste caso, o BC-PPO_Only1 vence o Brain LSTM em 91.3% das partidas. Este resultado mostra que é útil iniciar um treinamento PPO a partir de um modelo que foi treinado previamente com BC, porque o PPO acaba refinando o modelo aprendido anteriormente, ao invés de iniciar o aprendizado sem nenhum conhecimento preliminar (Veja Tabela 5.22).

Tabela 5.22: Resultado do T100 1vs1 BC+PPO_Only1 vs LSTM. Os participantes deste torneio foram os Brains $LSTM_2$ e BC+PPO_Only1. Note que para cada Brain existia apenas 1 agente

Resultado	**Número de Vitórias**
Empate	3 (0.3%)
$LSTM_2$	84 (8,4%)
BC+PPO_Only1	**913 (91,3%)**
Total	1000 (100%)

- **(T1000 1vs1 PPO_Only1 vs LSTM):** Para investigar se os resultados positivos do aprendizado com BC seguido do PPO eram devidos apenas ao PPO fizemos esse novo teste. Neste caso, nós não utilizamos o BC num treino prévio para descobrir se o PPO sozinho, que treinou controlando apenas 1 agente por cenário e que batalhava contra agentes que utilizavam o Brain LSTM, conseguiria aprender a derrotar o agente que usa LSTM. O agente PPO_Only1 venceu somente 13.88% das partidas contra o Brain LSTM. Estes resultados confirmam que um treinamento continuado com PPO não era o único responsável por fazer o BC+PPO_Only1 derrotar o Brain LSTM. Neste torneio, o Brain que treinou somente um agente por cenário (PPO_Only1) não foi capaz de derrotar o Brain LSTM (Veja Tabela 5.23).

Tabela 5.23: Resultado do T1000 1vs1 PPO_Only1 vs LSTM. Os participantes deste torneio foram os Brains $LSTM_2$ e PPO_Only1. Note que para cada Brain existia apenas 1 agente

Resultado	**Número de Vitórias**
Empate	165 (1.65%)
$\boldsymbol{LSTM_2}$	**8447 (84,47%)**
PPO_Only1	1388 (13,88%)
Total	10000 (100%)

- **(T1000 1vs1 BC+PPO_All vs LSTM):** Aqui, o Brain vencedor foi o BC+PPO_All com 56.69% das vitórias contra 39.98% do Brain LSTM. Estes resultado confirmam mais ainda que o BC providencia um bom ponto de partida para o PPO e o conhecimento adquirido num treinamento prévio com BC não foi esquecido (Veja Tabela 5.24).

Tabela 5.24: Resultado do T1000 1vs1 BC+PPO_All vs LSTM. Os participantes deste torneio foram os Brains $LSTM_2$ e BC+PPO_All. Note que para cada Brain existia apenas 1 agente

Resultado	**Número de Vitórias**
Empate	331 (3.31%)
$LSTM_2$	3999 (39,99%)
BC+PPO_All	**5670 (56,7%)**
Total	10000 (100%)

- **(T100 1vs1 BC+PPO_Only1 vs BC+PPO_All):** Também comparamos os dois Brains BC+PPO em um torneio T100 1x1, um contra o outro (veja Tabela 5.25). O BC+PPO_All venceu 49.5% das partidas enquanto que o BC+PPO_Only1 venceu 42.3% das partidas. Este resultado mostra que os agentes que aprendem simultaneamente em um mesmo cenário podem explorar mais o espaço de busca.

Tabela 5.25: Resultado do T100 1vs1 BC+PPO_Only1 vs BC+PPO_All. Os participantes deste torneio foram os Brains BC+PPO_Only1 e BC+PPO_All. Note que para cada Brain existia apenas 1 agente

Resultado	**Número de Vitórias**
Empate	82 (8.2%)
BC+PPO_Only1	423 (42,3%)
BC+PPO_All	**495 (49,5%)**
Total	1000 (100%)

Com a finalidade de comparar os dois modelos treinados com Brains BC+PPO, fizemos torneios T100 1vs1 em que cada Brain BC+PPO batalhou primeiro contra o Brain Híbrido e depois contra o Brain ZeroOuUm. Ambos os modelos treinados com BC+PPO perdem para estes dois Brains (veja as Tabelas 5.26, 5.27, 5.28, 5.29).

Tabela 5.26: Resultado do T100 1vs1 BC+PPO_Only1 vs H_1. Os participantes deste torneio foram os Brains BC+PPO_Only1 e H_1. Note que para cada Brain existia apenas 1 agente

Resultado	**Número de Vitórias**
Empate	62 (6.2%)
BC+PPO_Only1	433 (43,3%)
$\boldsymbol{H_1}$	**505 (50,5%)**
Total	1000 (100%)

Tabela 5.27: Resultado do T100 1vs1 BC+PPO_Only1 vs Z_1. Os participantes deste torneio foram os Brains BC+PPO_Only1 e Z_1. Note que para cada Brain existia apenas 1 agente

Resultado	**Número de Vitórias**
Empate	185 (18.5%)
BC+PPO_Only1	406 (40,6%)
$\boldsymbol{Z_1}$	**409 (40,9%)**
Total	1000 (100%)

Tabela 5.28: Resultado do T100 1vs1 BC+PPO_Only1 vs H_1. Os participantes deste torneio foram os Brains BC+PPO_Only1 e H_1. Note que para cada Brain existia apenas 1 agente

Resultado	**Número de Vitórias**
Empate	31 (3.1%)
BC+PPO_All	462 (46,2%)
$\boldsymbol{H_1}$	**507 (50,7%)**
Total	1000 (100%)

Tabela 5.29: Resultado do T100 1vs1 BC+PPO_All vs Z_1. Os participantes deste torneio foram os Brains BC+PPO_All e Z_1. Note que para cada Brain existia apenas 1 agente

Resultado	**Número de Vitórias**
Empate	42 (4,2%)
BC+PPO_All	375 (37,5%)
$\boldsymbol{Z_1}$	**583 (58,3%)**
Total	1000 (100%)

Os resultados apontam que o treinamento prévio de um modelo utilizando o algoritmo BC influencia um treinamento continuado posterior utilizando o algoritmo PPO, ao mesmo tempo em que é possível melhorar um agente refinando seu conhecimento adquirido com BC usando PPO. Os resultados dos últimos torneios contra o LSTM mostram que o Brain adversário usado no treinamento com BC torna-se um oponente mais fácil de ser derrotado em torneios subsequentes na fase de inferência. Pode ter ocorrido um *overfitting* do BC que acabou encapsulando o LSTM, porém, num torneio que fizemos entre o agente que utilizou apenas o BC contra o agente LSTM, o vencedor foi o LSTM que obteve um elevado número de vitórias. Acreditamos que, se diversificarmos os agentes oponentes na fase de treinamento do BC possamos gerar um modelo campeão real que vença mais oponentes. Diversificar os agentes oponentes, quer dizer que, todos os Brains vencedores, jogadores humanos, e outros métodos de AI poderiam ser usados para controlar o comportamento dos agentes nas partidas, que gravamos os arquivos de repetição, tais como métodos de busca e planejamento, e heurísticos. Ou seja, o jogador humano e especialista jogaria contra diferentes oponentes e gravaríamos todas as partidas. Em outro experimento, o especialista ao invés de ser um jogador humano poderia ser um agente inteligente ou as vezes ser um jogador humano e as vezes um agente.

Neste capítulo, no primeiro experimento descobrimos que o melhor tipo de representação de estado dentre as 5 analisadas nesta dissertação é a representação Híbrida porque gerou agentes mais vitoriosos. No segundo experimento, descobrimos que num treinamento Multi-Brain, a política de um Brain vai se sobrepor as outras atrapalhando a aprendizagem dos demais Brains, e que apesar de ganharmos tempo com este modo de treinamento (i.e. com 1 treinamento geramos 4 modelos, um para cada Brain, ao invés de gerar apenas 1 modelo como no treinamento Um-Brain), um treinamento Um-Brain gera modelos mais vitoriosos. Além disso, reforçamos que a melhor representação de estado é a Híbrida porque novamente neste experimento ela foi a mais vitoriosa. No

terceiro experimento, confirmamos que um treinamento Um-Brain utilizando PPO com LSTM e representação Híbrida gera agentes mais vitoriosos que um mesmo treinamento sem utilizar LSTM. No quarto experimento, descobrimos que um treinamento prévio com o método BC influencia positivamente um treinamento posterior com o PPO. Também descobrimos que agentes treinados com este método se tornam especializados em derrotar o oponente com que treinaram na fase do treinamento prévio com o BC. Por último, verificamos que os agentes BC+PPO mesmo vencendo os agentes LSTM podem perder para agentes diferentes dos oponentes vistos em seu treinamento com o BC.

Capítulo 6

Conclusão

Neste trabalho propusemos um ambiente de aprendizagem Bomberman (BLE) utilizando algoritmos de aprendizagem por Reforço e por Imitação, usando o conjunto de ferramentas ML-Agents. O BLE contém 5 estilos de representação de estado diferentes, todas assumindo um espaço de representação em vetores de características, mas diferindo nas formas como cada célula da grade do jogo será enviada a este vetor, que pode influenciar o que os agentes aprendem. Realizamos uma série de experimentos considerando as diferentes representações de estado, treinando com um Brain e múltiplos Brains. Também treinamos com dois métodos diferenciados de rede neural (MLP e LSTM). Finalmente testamos uma aprendizagem com o BC seguida pelo PPO. Os resultados aqui conduzidos mostraram que o treinamento Um-Brain produz melhores agentes do que o treinamento Multi-Brain. Também mostramos que a representação Híbrida alcança os melhores resultados em comparação com as outras formas de representação. Além disso, os resultados apontam que o acoplamento do LSTM dentro do algoritmo PPO produz agentes mais inteligentes, e que o treinamento prévio com o algoritmo BC pode influenciar o treinamento subsequente com o PPO.

6.1 Técnicas Geradas

Nessa pesquisa, em nosso fork do ML-Agents v0.5, foi desenvolvida uma forma de acelerar o treinamento via BC implementado no ML-Agents. Com o ML-Agents v0.5, para se demonstrar um comportamento que será aprendido por um agente aprendiz, o treinamento precisava ser rodado no modo normal de execução para capturar ações de um jogador especialista através de um Brain do tipo Player. Como utilizamos um Brain Heurístico para ler os movimentos de um arquivo de repetição, essa restrição de executar o treinamento

no modo normal foi superada. Também desenvolvemos 2 novos tipos de representação de estado que obtiveram um bom desempenho nos experimentos: as representações Híbrida e a ZeroOuUm. Além disso, desenvolvemos um método para se carregar um modelo do TensorFlow gerado num treinamento X para se iniciar um treinamento Y. Em resumo, todos os tensores que existem em ambos modelos são carregados ao se iniciar um novo treinamento. Caso os tensores sejam diferentes, estes são apenas inicializados normalmente. Por meio deste método, foi possível iniciar o treinamento do PPO com um modelo já treinado com o método BC.

6.2 Limitações

Em algumas situações os modelos treinados não sabem como se comportar contra inimigos passivos por exemplo. Isto acontece porque eles não viram esses inimigos durante o treinamento (e.g. inimigos passivos que raramente se movem podem causar comportamentos estranhos). Quando o espaço e o tempo não são discretizados, os algoritmos de aprendizagem presentes no ML-Agents têm dificuldades em aprender a jogar o BLE, porque a recompensa recebida ao destruir um bloco ou um inimigo demora várias iterações para ser dada (recompensa em atraso). Outra limitação, foi o fato dos movimentos do especialista gravados no arquivo de repetição terem sido capturados de apenas um jogador, e os inimigos utilizaram apenas um Brain ($LSTM_2$).

De um modo geral, quando um aprendiz está começando a aprender determinada tarefa, caso ele receba do ambiente ou mentor muito mais reforços negativos do que positivos, provavelmente ele desistirá de aprender tal tarefa. Durante este trabalho, houve uma situação semelhante, em que o aprendiz (o agente inteligente) desistia de aprender a colocar bombas e a usá-las de uma forma coerente para destruir blocos e inimigos no cenário. Isso acontecia porque a recompensa negativa relativa a morte por bomba era tão alta quanto a recompensa positiva dada por eliminar algum oponente.

6.3 Trabalhos Futuros

Como trabalho futuro pretendemos melhorar o ambiente BLE dando ao agente a capacidade de colocar várias bombas de uma só vez e adquirir novos poderes, ter cenários de tamanho variável, ter outras métricas para compararmos os agentes (atualmente utilizamos torneios), adicionar modo de jogo cooperativo com times e testar outros algoritmos

de RL e IL. Além disso, podemos treinar um Brain que além de treinar jogando contra si mesmo, também treine jogando contra variados oponentes e inteligências. Isso aumentaria a generalização dos algoritmos PPO. Podemos diversificar também os oponentes que são usados para treinar um agente com o método BC e também diversificar o especialistas utilizados (e.g. um agente PPO pode ser um especialista). Nós também poderíamos utilizar os algoritmos e representações utilizados nesta dissertação para desenvolver agentes para competir no ambiente Pommerman[41]. Também, alguns agentes heurísticos *hard coded*, que utilizam Brains heurísticos, poderiam ser desenvolvidos para batalhar contra os agentes que utilizam RL e IL. Com isso, poderíamos comparar diferentes tipos de agentes entre si.

No BLE, utilizamos redes neurais como aproximação não linear de função, para gerar os valores de cada estado do jogo (o crítico do PPO). A critério de comparação, poderíamos ter utilizado alguma função linear como aproximação de função.

Percebemos nesta dissertação, que o tipo de representação utilizado pelos algoritmos que utilizam redes neurais influenciou o treinamento e a aprendizagem dos modelos gerados. Uma pesquisa mais detalhada e aprofundada poderia ser realizada para comparar as representações e descobrir por exemplo, uma representação pode combinar melhor com um método do que com outro? Por que a representação *One-Hot* é a recomendada, para representar dados categóricos, pelos pesquisadores que utilizam redes neurais? Este estudo sobre as representações pode gerar um grande impacto.

6.4 Considerações Finais

Para finalizar, nessa dissertação criamos novas formas de representação de estado que se mostraram superiores à forma utilizada em [21] tais como as representações Híbrida e ZeroOuUm. Por meio de vários experimentos, comprovamos que a representação de estado escolhida para alimentar uma rede neural do PPO tem a capacidade de influenciar toda a aprendizagem deste algoritmo. Investigamos também algoritmos de Aprendizagem por Reforço e por Imitação que utilizam redes neurais artificiais para jogar o Bomberman. Criamos um novo método em que aproveitamos um treinamento prévio realizado com o BC para iniciar a rede neural de um treinamento posterior com o PPO. Com esta nova técnica, notamos que podemos influenciar um treinamento posterior com o PPO, porque o oponente que foi utilizado no treinamento do BC se tornou um oponente fácil de ser derrotado no treinamento posterior. Confirmamos que utilizar PPO com uma rede neural

mais efetiva em tarefas sequenciais, como a LSTM, produz agentes mais vitoriosos que o PPO sozinho.

Acreditamos que métodos que utilizem Aprendizagem por Imitação em conjunto com métodos de Aprendizagem por Reforço terão um grande impacto num futuro próximo. Essa combinação de métodos tem a capacidade de resolver muitos problemas e jogos.

Referências

[1] Abbeel, P.; Ng, A. Y. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning* (2004), ACM.

[2] Argall, B. D.; Chernova, S.; Veloso, M.; Browning, B. A survey of robot learning from demonstration. *Robotics and autonomous systems 57*, 5 (2009), 469–483.

[3] Cho, K.; Van Merriënboer, B.; Gulcehre, C.; Bahdanau, D.; Bougares, F.; Schwenk, H.; Bengio, Y. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078* (2014).

[4] Clark, J.; Amodei, D. Faulty reward functions in the wild. `https://blog.openai.com/faulty-reward-functions/`, 2016. Accessed in 2018-05-02.

[5] da Cruz Lopes, M. A. Bomberman as an artificial intelligence platform. Master's thesis, Universidade do Porto, 2016.

[6] DeepMind, G. Alphastar: Mastering the real-time strategy game starcraft ii. `https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii`, 2019.

[7] Eli Karasik, A. H. Intro to ai bomberman. urlhttp://www.cs.huji.ac.il/ ai/projects/2012/Bomberman/, 2013. Accessed in 2018-11-25.

[8] Eric Van de Kerckhove, B. B. How to make a game like bomberman with unity. urlhttps://www.raywenderlich.com/244-how-to-make-a-game-like-bomberman-with-unity, 2018. Accessed in 2018-11-28.

[9] Goodfellow, I.; Bengio, Y.; Courville, A.; Bengio, Y. *Deep learning*, vol. 1. MIT press Cambridge, 2016.

[10] Guez, A.; Weber, T.; Antonoglou, I.; Simonyan, K.; Vinyals, O.; Wierstra, D.; Munos, R.; Silver, D. Learning to search with mctsnets. *arXiv preprint arXiv:1802.04697* (2018).

[11] Haas, J. K. A history of the unity game engine. Interactive Qualifying Project, Worcester Polytechnic Institute, Worcester, Massachusetts 2014.

[12] Hester, T.; Vecerik, M.; Pietquin, O.; Lanctot, M.; Schaul, T.; Piot, B.; Horgan, D.; Quan, J.; Sendonaris, A.; Osband, I., et al. Deep q-learning from demonstrations. In *Thirty-Second AAAI Conference on Artificial Intelligence* (2018), pp. 3223–3230.

[13] Ho, J.; Gupta, J.; Ermon, S. Model-free imitation learning with policy optimization. In *International Conference on Machine Learning* (2016), pp. 2760–2769.

[14] Hochreiter, S.; Schmidhuber, J. Long short-term memory. *Neural computation 9*, 8 (1997), 1735–1780.

[15] Hsu, F.-h. Ibm's deep blue chess grandmaster chips. *IEEE Micro 19*, 2 (1999), 70–81.

[16] Hussein, A.; Gaber, M. M.; Elyan, E.; Jayne, C. Imitation learning: A survey of learning methods. *ACM Computing Surveys (CSUR) 50*, 2 (2017), 21.

[17] Irpan, A. Deep reinforcement learning doesn't work yet. urlhttps://www.alexirpan.com/2018/02/14/rl-hard.html, 2018. Accessed in 2018-05-02.

[18] Juliani, A.; Berges, V.-P.; Vckay, E.; Gao, Y.; Henry, H.; Mattar, M.; Lange, D. Unity: A general platform for intelligent agents. *arXiv preprint arXiv:1809.02627* (2018).

[19] Kartal, B.; Hernandez-Leal, P.; Gao, C.; Taylor, M. E. Safer deep rl with shallow mcts: A case study in pommerman. *arXiv preprint arXiv:1904.05759* (2019).

[20] Kishimoto, A. Inteligência artificial em jogos eletrônicos. *Academic research about Artificial Intelligence for games* (2004).

[21] Kormelink, J. G.; Drugan, M. M.; Wiering, M. A. Exploration methods for connectionist q-learning in bomberman. In *ICAART (2)* (2018), pp. 355–362.

[22] Laird, J. E. Research in human-level ai using computer games. *Commun. ACM 45*, 1 (Jan. 2002), 32–35.

[23] Lanham, M. *Learn Unity ML-Agents–Fundamentals of Unity Machine Learning: Incorporate new powerful ML algorithms such as Deep Reinforcement Learning for games*. Packt Publishing Ltd, 2018.

[24] Lapan, M. *Deep Reinforcement Learning Hands-On: Apply modern RL methods, with deep Q-networks, value iteration, policy gradients, TRPO, AlphaGo Zero and more*. Packt Publishing Ltd, 2018.

[25] Li, Y. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274* (2017).

[26] Lowe, R.; Wu, Y.; Tamar, A.; Harb, J.; Abbeel, O. P.; Mordatch, I. Multi-agent actor-critic for mixed cooperative-competitive environments. In *Advances in Neural Information Processing Systems* (2017), pp. 6379–6390.

[27] Malysheva, A.; Sung, T. T.; Sohn, C.-B.; Kudenko, D.; Shpilman, A. Deep multi-agent reinforcement learning with relevance graphs. *arXiv preprint arXiv:1811.12557* (2018).

[28] Mendonça, P. G. B. S. d.; Souza, V. S. d. Uma implementação de um agente autônomo para o jogo bomberman com aprendizado por reforço. Final Project (Bachelor in Computer Science), UFF (Universidade Federal Fluminense), Niteroi, Brazil 2018.

[29] Mnih, V.; Badia, A. P.; Mirza, M.; Graves, A.; Lillicrap, T.; Harley, T.; Silver, D.; Kavukcuoglu, K. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning* (2016), pp. 1928–1937.

[30] Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; Riedmiller, M. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[31] Monteiro, J.; Granada, R.; Pinto, R. C.; Barros, R. C. Beating bomberman with artificial intelligence. In *Anais do XV Encontro Nacional de Inteligência Artificial e Computacional* (2018), SBC, pp. 353–364.

[32] Ng, A. Y.; Russell, S. J. Algorithms for inverse reinforcement learning. In *Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000), Stanford University, Stanford, CA, USA, June 29 - July 2, 2000* (2000), P. Langley, Ed., Morgan Kaufmann, pp. 663–670.

[33] Nielsen, M. A. *Neural networks and deep learning*, vol. 25. Determination press San Francisco, CA, USA:, 2015.

[34] Olah, C. Understanding lstm networks. `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`, 2018.

[35] OpenAI. Openai dota 2. `https://blog.openai.com/dota-2`, 2018.

[36] OpenAI. Openai five. `https://blog.openai.com/openai-five/`, 2018.

[37] Osogami, T.; Takahashi, T. Real-time tree search with pessimistic scenarios. *arXiv preprint arXiv:1902.10870* (2019).

[38] Pathak, D.; Agrawal, P.; Efros, A. A.; Darrell, T. Curiosity-driven exploration by self-supervised prediction. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017* (2017), D. Precup and Y. W. Teh, Eds., vol. 70 of *Proceedings of Machine Learning Research*, PMLR, pp. 2778–2787.

[39] Pellegrini, J.; Wainer, J. Processos de decisão de markov: um tutorial. *Revista de Informática Teórica e Aplicada 14*, 2 (2007), 133–179.

[40] Pomerleau, D. Efficient training of artificial neural networks for autonomous navigation. *Neural Computation 3*, 1 (January 1991), 88–97.

[41] Resnick, C.; Eldridge, W.; Ha, D.; Britz, D.; Foerster, J.; Togelius, J.; Cho, K.; Bruna, J. Pommerman: A multi-agent playground. *arXiv preprint arXiv:1809.07124* (2018).

[42] Russell, S. J.; Norvig, P. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.

[43] Schmidhuber, J. Deep learning in neural networks: An overview. *Neural networks 61* (2015), 85–117.

[44] Schulman, J.; Levine, S.; Abbeel, P.; Jordan, M.; Moritz, P. Trust region policy optimization. In *International Conference on Machine Learning* (2015), pp. 1889–1897.

[45] Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; Klimov, O. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[46] Shah, D.; Singh, N.; Talegaonkar, C. Multi-agent strategies for pommerman.

[47] Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; Van Den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M., et al. Mastering the game of go with deep neural networks and tree search. *Nature 529*, 7587 (2016), 484.

[48] Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; Lillicrap, T.; Simonyan, K.; Hassabis, D. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science 362*, 6419 (2018), 1140–1144.

[49] Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; Huang, A.; Guez, A.; Hubert, T.; Baker, L.; Lai, M.; Bolton, A., et al. Mastering the game of go without human knowledge. *Nature 550*, 7676 (2017), 354.

[50] Soft, H. Bomberman. `https://en.wikipedia.org/wiki/Bomberman`, 1983. Accessed in 2018-08-28.

[51] Sutton, R. S.; Barto, A. G. *Reinforcement learning: An introduction*, 2 ed. MIT press, 2018.

[52] Unity3D. Ml-agents. `https://github.com/Unity-Technologies/ml-agents`, 2018. Accessed in 2018-08-23.

[53] Unity3D. Training with proximal policy optimization. `https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-PPO.md`, 2018. Accessed in 2018-08-23.

[54] Unity3D. Unity 3d. `https://unity3d.com`, 2018. Accessed in 2018-05-02.

[55] Zhan, E.; Zheng, S.; Yue, Y.; Lucey, P. Generative multi-agent behavioral cloning. *arXiv preprint arXiv:1803.07612* (2018).

[56] Zhou, H.; Gong, Y.; Mugrai, L.; Khalifa, A.; Nealen, A.; Togelius, J. A hybrid search agent in pommerman. In *Proceedings of the 13th International Conference on the Foundations of Digital Games* (2018), ACM, p. 46.

APÊNDICE A – Hiper-parâmetros do PPO e BC

Os autores do plugin ML-agents [52], que foi utilizado nessa pesquisa, explicam na documentação do plugin cada hiper-parâmetros. Abaixo, por meio da Tabela A.1, explicamos resumidamente os hiper-parâmetros e informamos se cada um é utilizado ou não em ambos algoritmos. Há informações erradas na documentação do plugin quanto ao uso dos hiper-parâmetros em cada algoritmo. Informamos através de um asterisco estes hiper-parâmetros (*).

Uma explicação mais detalhada dos hiper-parâmetros, que contém as melhores práticas para ajustar o processo de treinamento quando os parâmetros padrões não parecem dar o nível de desempenho desejado, pode ser encontrada aqui [53].

Tabela A.1: Resumo de Todos Hiper-parâmetros do PPO e do BC

Nome	Descrição	Algoritmos
batch_size	O número de experiências em cada iteração do gradiente descendente.	PPO, BC
batches_per_epoch	Na aprendizagem por imitação, o número de lotes de exemplos de treinamento para coletar antes de treinar o modelo.	BC
*beta	A força da regularização da entropia.	PPO
brain_to_imitate	Para aprendizagem por imitação, o nome do GameObject que contém o componente do Cérebro para imitar.	BC
buffer_size	O número de experiências a coletar antes de atualizar o modelo da política.	PPO
curiosity_enc_size	O tamanho da codificação a ser usada nos modelos de avanço e inverso no módulo Curiosity.	PPO
curiosity_strength	Magnitude da recompensa intrínseca gerada pelo Intrinsic Curiosity Module.	PPO
*epsilon	Influencia a rapidez com que a política pode evoluir durante o treinamento.	PPO
gamma	A taxa de desconto de recompensa para a Generalized Advantage Estimator (GAE).	PPO
hidden_units	O número de unidades nas camadas ocultas da rede neural..	PPO, BC
lambd	O parâmetro de regularização.	PPO
learning_rate	A taxa inicial de aprendizado para gradiente descendente.	PPO, BC
max_steps	O número máximo de etapas de simulação a serem executadas durante uma sessão de treinamento.	PPO, BC
memory_size	O tamanho da memória que um agente deve manter. Usado para treinamento com uma rede neural recorrente.	PPO, BC
*normalize	Se deve normalizar automaticamente as observações.	PPO
*num_epoch	O número de passos a serem feitos através do buffer de experiência ao executar a otimização de descida de gradiente.	PPO
num_layers	O número de camadas ocultas na rede neural.	PPO, BC
sequence_length	Define o tamanho que as sequências de experiências devem ter durante o treinamento. Utilizado apenas para treinamento com uma rede neural recorrente.	PPO, BC
summary_freq	Com que frequência, em passos, para salvar estatísticas de treinamento. Isso determina o número de pontos de dados mostrados pelo TensorBoard.	PPO, BC
time_horizon	Quantos passos de experiência para coletar por agente antes de adicioná-lo ao buffer de experiência.	PPO, BC
trainer	O tipo de treinamento a ser executado: "ppo", "imitation"ou "imitationCustom"(nossa autoria).	PPO, BC
use_curiosity	Treinar usando um sinal de recompensa intrínseco adicional gerado a partir do Intrinsic Curiosity Module.	PPO
use_recurrent	Treinar usando uma rede neural recorrente.	PPO, BC

APÊNDICE B – Hammerman

Após baixar o projeto do Bomberman, para efetivar o ambiente Hammerman, é necessário que seja efetuada uma troca de ramo no Git. O ramo do Hammerman foi chamado de **hammerman_experiment**. O comando do git usado para trocar de ramo é mostrado abaixo.

```
git checkout hammerman_experiment
```

Hammerman foi o nome dado para o ambiente do Bomberman desenvolvido nesta dissertação e que não possui bombas, onde o agente não possui a ação de colocar bomba, mas pode atacar com um martelo a um de distância nas quatro direções: cima, baixo, esquerda e direita. O bloco atingido pelo martelo é destruído na mesma iteração. Por causa disso, não há atraso de recompensa (reward delayed) nesse ambiente. A meta do agente nesse ambiente é alcançar o bloco objetivo. E para isso, o agente deve abrir caminho para chegar até a esse bloco.

Neste protótipo, o cenário do jogo tem uma dimensão de sete linhas por oito colunas. O agente é iniciado na primeira posição (Esquerda-Topo) e tem a meta de alcançar o objetivo que está localizado na última posição (Direita-Baixo). Veja a Figura B.1. Note que as células que circundam o cenário não são utilizadas na representação do grid nesse protótipo. Caso contrário seriam nove linhas por dez colunas.

Observação: para o protótipo do Hammerman não há bomba, perigo, fogo, oponente. Ao invés disso há o bloco objetivo.

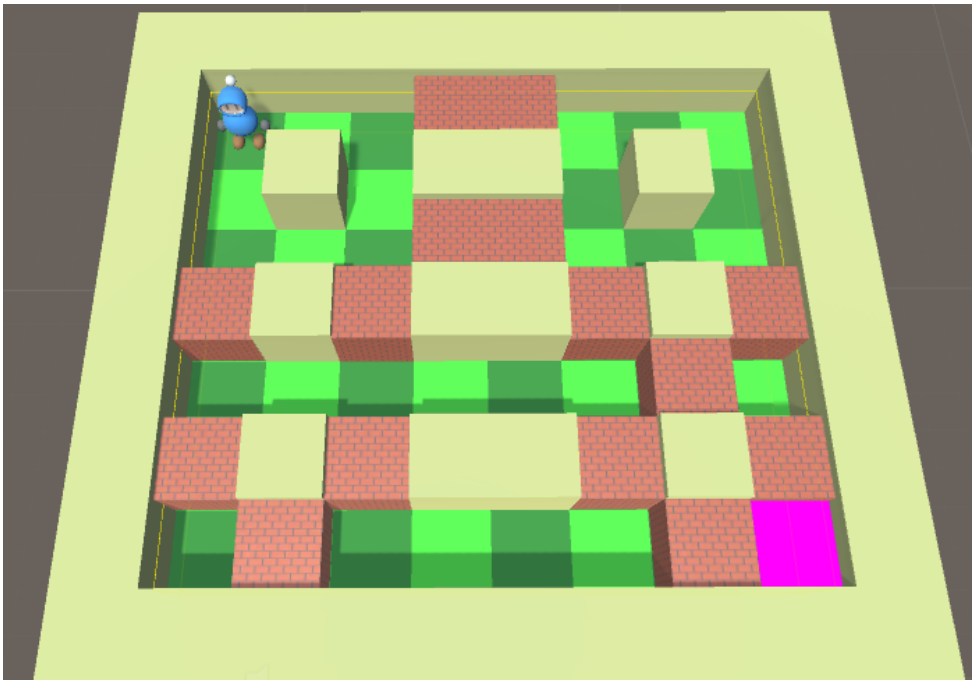

Figura B.1: Cenário do protótipo Hammerman 7x8. O agente tem a cor azul. O objetivo tem a cor rosa. Os blocos com cor de tijolo são destrutíveis.

Em cada iteração do jogo, o agente pode escolher uma entre nove ações. A Tabela B.1 mostra as possíveis ações. As ações com martelo destroem o bloco caso haja um bloco na direção selecionada.

Tabela B.1: Ações Possíveis para o Hammerman

Ação	**Descrição**
0	Ficar parado
1	Ir pra cima
2	Ir pra baixo
3	Ir pra esquerda
4	Ir pra direita
5	Martelo pra cima
6	Martelo pra baixo
7	Martelo pra esquerda
8	Martelo pra direita

Após executar ações no ambiente, o agente recebe recompensas de acordo com o novo estado do jogo ou com a ação escolhida. Neste protótipo, as recompensas foram

distribuídas de acordo com a Tabela B.2. Em cada iteração, uma ou mais recompensas podem ser dadas ao agente. Ou seja, a recompensa é adicionada à recompensa acumulada da iteração atual do agente e não sobrescrita.

Tabela B.2: Recompensas para o Hammerman

Estado/Ação	**Recompensa**
Alcançar o objetivo	1.0
Destruir bloco	0.51
Melhor Recorde de aproximação do objetivo	0.51
Aproximar-se do objetivo	0.007625
Distanciar-se do objetivo	-0.007625
Penalidade por iteração	-0.001f
Escolher ação de ficar parado	-0.03
Tentar andar para posição inválida	-0.02
Andar para posição válida	0.02
Tentar martelar posição sem bloco destrutível	-0.01
Martelar bloco destrutível	0.02
Atingir limite de iterações por episódio	-1.0

APÊNDICE C – Informações Adicionais Sobre os Experimentos no BLE

Neste Apêndice C, adicionamos todos os dados extras sobre cada um dos experimentos que foram realizados no BLE no Capítulo 5. Os gráficos sobre as estatísticas que não foram mostrados lá, estão disponíveis aqui. Organizamos as seções deste apêndice como as seções dos 3 primeiros experimentos que foram analisados no Capítulo 5. Como não há informações adicionais do último experimento, relativo ao BC+PPO, não foi adicionada uma seção deste experimento aqui.

C.1 Escolhendo a representação de estado para o jogo Bomberman de forma ingênua

Nesta seção, primeiramente mostramos os gráficos de cada uma das estatísticas que não foram apresentados no Capítulo 5, Seção 5.2. Depois, mostramos um primeiro experimento que fizemos onde escolhemos os modelos de forma ingênua que seriam utilizados por cada Brain (e seu tipo de representação). Dizemos que a escolha foi feita de forma ingênua, porque escolhemos apenas o modelo, para cada tipo de representação, que obteve a maior recompensa acumulada durante os treinamentos. Por exemplo, dos 5 modelos gerados para representação Híbrida, escolhemos o modelo que obteve a maior recompensa acumulada, e este modelo participou do T1000. Isso foi feito para cada tipo de representação.

Quando comparamos a entropia dos modelos gerados pelos treinamentos nas Figuras C.1 e C.2, percebemos que ela se comportou da forma esperada (diminuiu consistentemente durante o treinamento). Apenas as entropias do Brain de Flag Binária e de Flag Binária Normalizada destoaram mais das outras até a iteração 501K, e, após essa iteração, apenas o Brain de Flag Binária continuou destoado dos outros Brains por todo

o treinamento. Além disso, ele possuiu, em média, 6 centésimos a mais do valor das demais entropias no término do treinamento. Foi constatado também que as entropias dos Brains ICAART, Híbrido, ZeroOuUm e Flag Binária Normalizada são praticamente, em média, as mesmas desde a iteração 501K até ao término do treinamento. Na ordem da menor entropia para a maior, a classificação ficou da seguinte forma: ICAART, Híbrido, ZeroOuUm, Flag Binária Normalizada e Flag Binária.

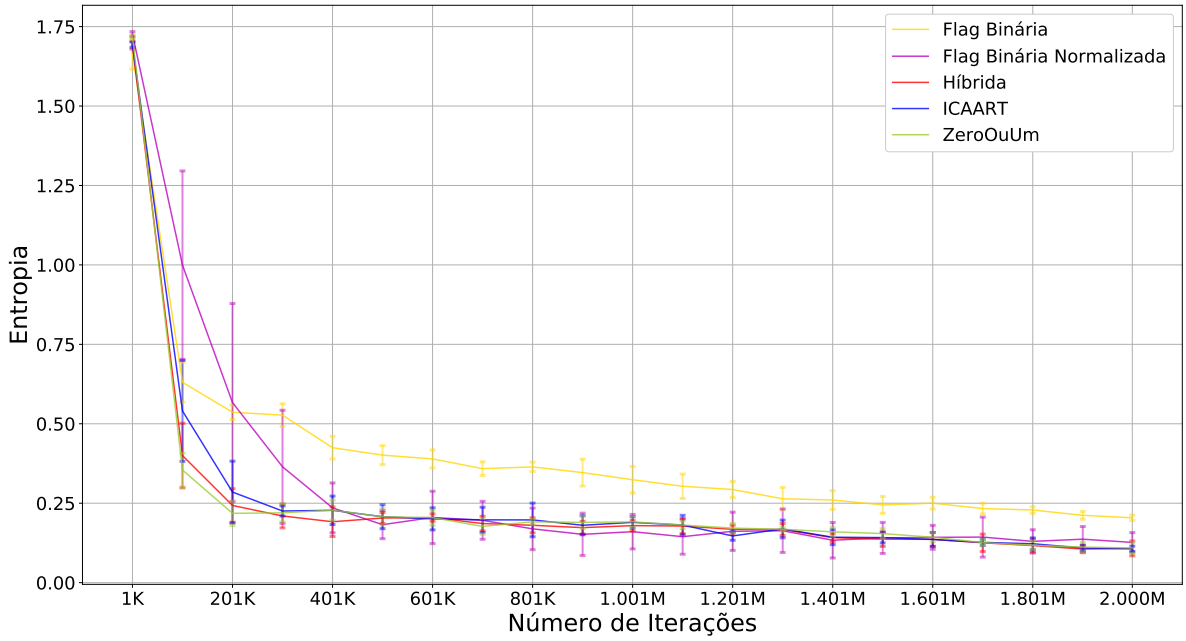

Figura C.1: Gráfico de Entropia obtido nos treinamentos no ambiente Bomberman comparando os diferentes tipos de representação do estado. Desvio padrão é mostrado por linhas verticais.

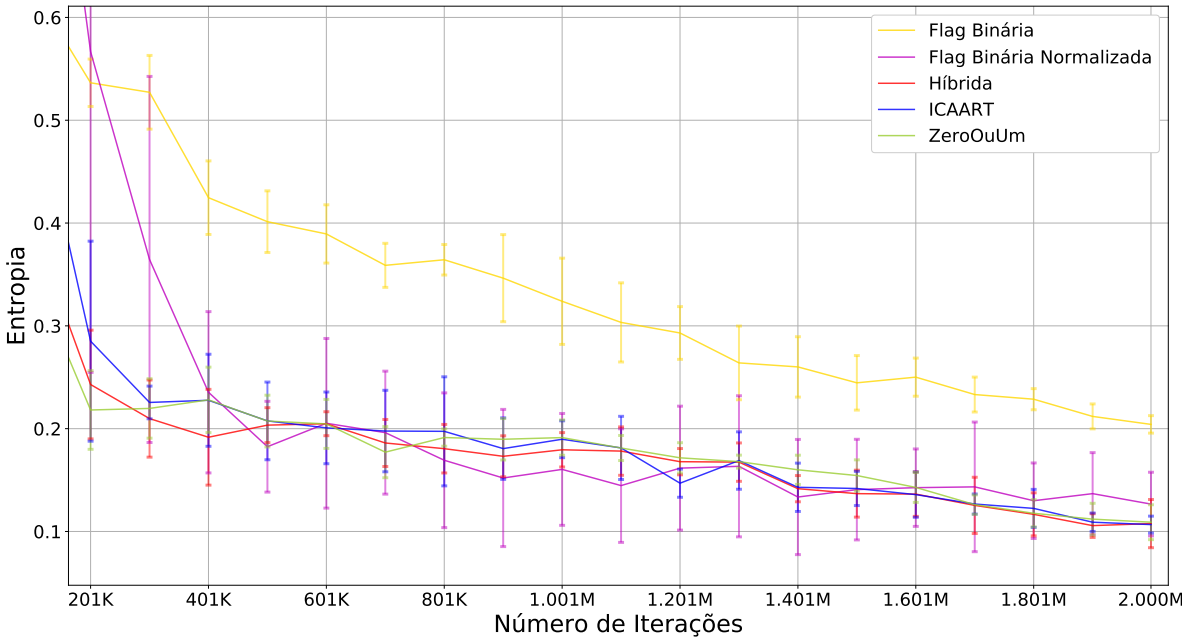

Figura C.2: Detalhamento de um trecho do Gráfico de Entropia obtido nos treinamentos no ambiente Bomberman comparando os diferentes tipos de representação do estado. Desvio padrão é mostrado por linhas verticais.

Quanto ao tamanho dos episódios, representado no gráfico da Figura C.3, observamos que os Brains ZeroOuUm, Híbrido e ICAART foram os Brains que obtiveram os melhores resultados respectivamente, possuindo uma tendência de crescimento até o final do treinamento. O Brain de Flag Binária e o de Flag Binária Normalizada tiveram o pior desempenho nessa estatística e não conseguiram se igualar a nenhum dos outros Brains, nem mesmo considerando o desvio padrão. Vale salientar que considerando o desvio padrão, ICAART, ZeroOuUm e Híbrido também ficaram empatados. Note que, no início do treinamento, sobreviver por mais tempo num episódio não quer dizer muita coisa, já que o agente simplesmente pode estar isolado na sua posição inicial e não abriu caminho até os seus oponentes, utilizando bombas. Além disso, note que no início do treinamento os agentes tendem a morrer mais rapidamente, porque estão aprendendo a utilizar a bomba, descobrindo suas consequências. Outra curiosidade que pode ser observada é que o Brain de Flag Binária Normalizada, até a iteração 1.401M, obteve um desempenho próximo ao do Brain ICAART, mas no final do treinamento ficou em penúltimo lugar. Esse comportamento talvez se deve à possibilidade de o agente não ter aprendido a utilizar bombas efetivamente (fugir de bombas e colocar bombas para destruir blocos e inimigos).

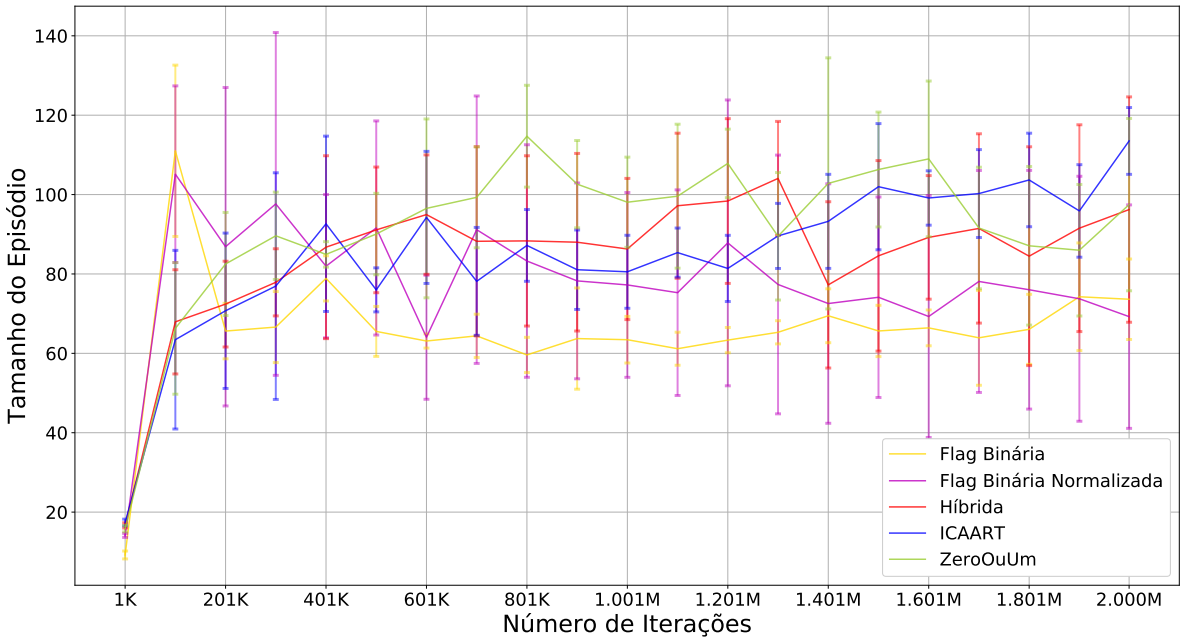

Figura C.3: Gráfico do Tamanho do Episódios obtido nos treinamentos no ambiente Bomberman comparando os diferentes tipos de representação do estado. Desvio padrão é mostrado por linhas verticais.

Quando a estimativa de valor é analisada, conforme o gráfico da Figuras C.4 e C.5, observamos que o resultado obtido não se comportou conforme o que foi dito anteriormente sobre o comportamento ideal dessa estatística. Os valores não foram aumentando do início até o final do treinamento para os Brains Híbrido, ICAART e ZeroOuUm, mas tiveram um comportamento alto nas 101K primeiras iterações. O único Brain que obteve uma subida até o final do treinamento, desde a iteração 101K, foi o de Flag Binária Normalizada. O Brain de Flag Binária teve uma certa tendência de subida, mas no final do treinamento começou a ter uma tendência de descida. Os outros Brains tiveram uma tendência de descida após a iteração 101K até o final do treinamento. Os Brain que obtiveram o melhor desempenho nessa estatística do melhor para o pior foram: Flag Binária, Flag Binária Normalizada, ICAART, Híbrido e ZeroOuUm.

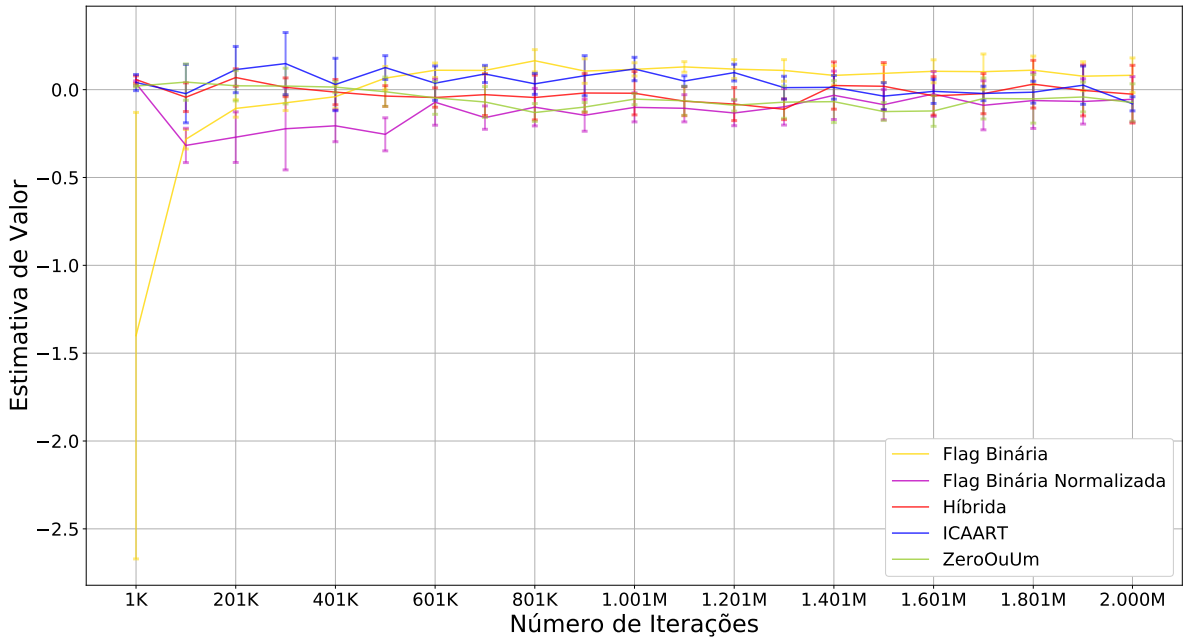

Figura C.4: Gráfico de Estimativa de Valor obtido nos treinamentos no ambiente Bomberman comparando os diferentes tipos de representação do estado. Desvio padrão é mostrado por linhas verticais.

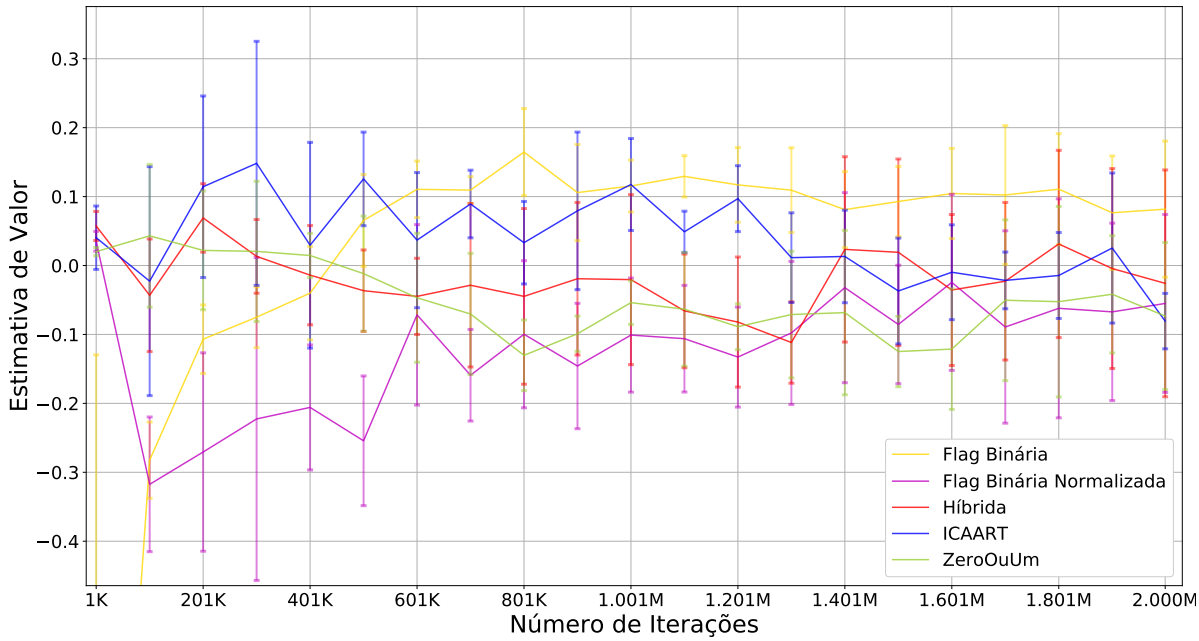

Figura C.5: Gráfico de Estimativa de Valor com detalhamento obtido nos treinamentos no ambiente Bomberman comparando os diferentes tipos de representação do estado. Desvio padrão é mostrado por linhas verticais.

No gráfico de Loss de Valor da Figura C.6 observamos que os valores iniciais do Brain

de Flag Binária destoam muito dos valores iniciais dos outros Brains. Além disso, após a iteração 401K, seus valores voltam a destoar dos demais. Se considerarmos o início do gráfico a partir da iteração 101K, observamos que os únicos Brains que não tiveram o comportamento adequado foram os de Flag Binária e Flag Binária Normalizada, porque eles não pararam de crescer, já que suas recompensas acumuladas também não pararam de crescer. Já o Loss de valor dos outros Brains cresceram no início do treinamento e quando a recompensa acumulada se estabilizou, os valores do loss de valor iniciaram a descida no gráfico.

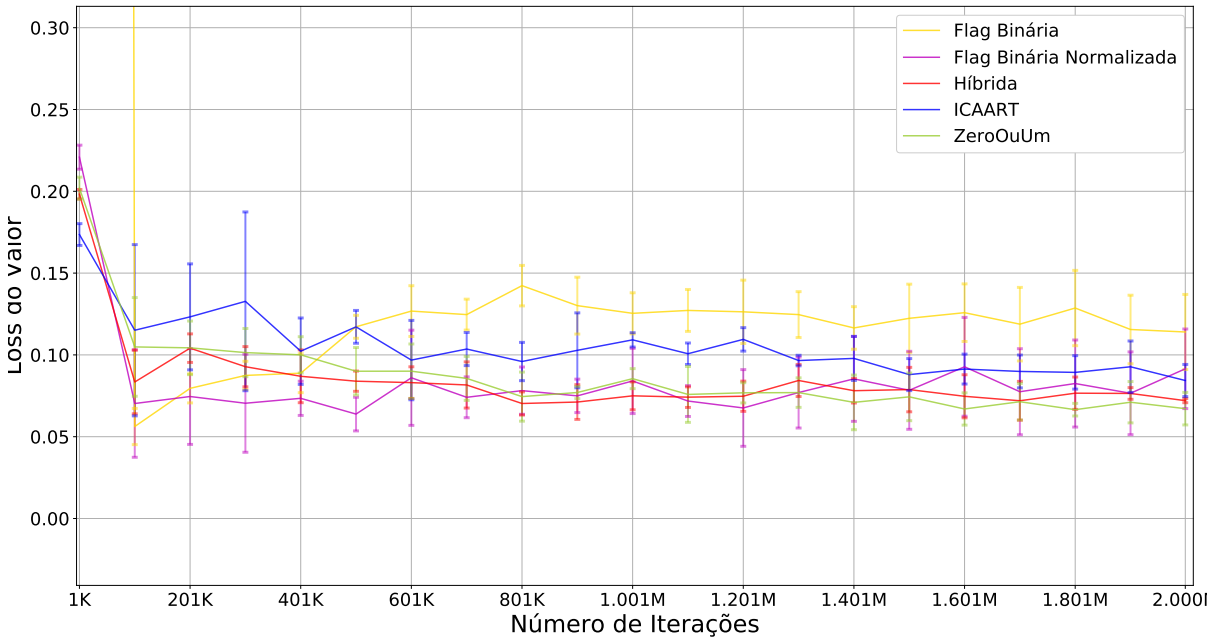

Figura C.6: Gráfico de Loss de Valor obtido nos treinamentos no ambiente Bomberman comparando os diferentes tipos de representação do estado. Desvio padrão é mostrado por linhas verticais.

No gráfico da Figura C.7 observamos que a magnitude do Loss de Política de todos os Brains possuem uma tendência de queda e um comportamento esperado.

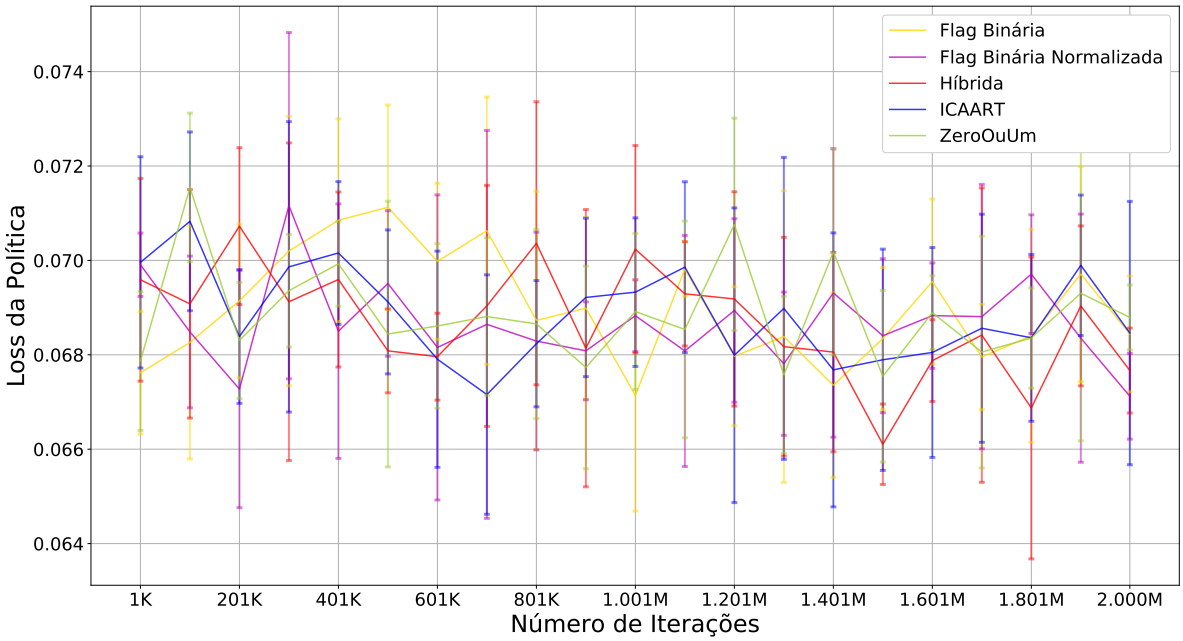

Figura C.7: Gráfico de Loss de Política obtido nos treinamentos no ambiente Bomberman comparando os diferentes tipos de representação do estado.

Após os treinamentos, configuramos um torneio, **T1000 Max Recompensa Acumulada**, onde, para cada tipo de representação foi escolhido o modelo gerado pelo treinamento dos Brains que utilizaram esta representação (dentre os cinco) que obteve a melhor recompensa acumulada. Configuramos para cada cenário do BLE a geração fixa de 4 agentes em todo episódio. Cada agente está atrelado a um Brain. Os Brains participantes deste torneio foram os integrantes do conjunto: $conj\_max\_recompensa = \{FB_5,\ H_2,\ I_4\ e\ Z_1\}$.

Note que nessa fase de batalha entre os agentes/Brains, não foi utilizado o Brain FBN_4, porque há apenas 4 vagas no máximo para os Brains participantes de um torneio, e por isso, fizemos um torneio prévio entre FB_5 e FBN_4, que foram as representações de estado que obtiveram as piores recompensas acumuladas durante os treinamentos, em que o primeiro Brain se saiu vitorioso (ver Tabela C.1). Além disso, é bom salientar que o desempenho destes dois Brains, na prática, foi inferior ao de todos os outros, considerando as estatísticas exibidas nos gráficos. Após a configuração dos Brains e seus respectivos modelos, em fase de inferência, o ambiente é executado para gerar a estatística de qual agente/Brain venceu mais partidas dentre o T1000 para cada um dos 10 cenários. Os resultados parciais deste torneio são mostrados nas Tabelas C.2 e C.3, onde foram divididos em cenários estáticos e aleatórios respectivamente. Já o resultado Total, que contém tanto os resultados nos cenários estáticos quanto os dos cenários aleatórios, é apresentado na Tabela C.4.

Tabela C.1: Resultado do T100 prévio entre Flag Binária e Flag Binária Normalizada. Total e porcentagem de Vitórias considerando todos cenários. Os participantes deste torneio foram os Brains F_5 e FBN_4. Note que para cada Brain haviam dois agentes

Resultado	**Número de Vitórias**
Empate	97 (9,7%)
$\boldsymbol{F_5}$	**519 (51,9%)**
FBN_4	384 (38,4%)
Total	1000 (100%)

Na Tabela C.2, que corresponde aos resultados das batalhas nos cenários estáticos do **T1000 Max Recompensa Acumulada** que foi composto pelo conjunto de Brains: FB_5, H_2, I_4 *e* Z_1, foi observado que o Brain com representação ZeroOuUm obteve o maior número de vitórias. Em segundo lugar, ficou o ICAART que usou o quarto modelo obtido nos seus treinamentos. Note que o desempenho do Brain com Flag Binária foi pior até mesmo do que o número de empates, e que o Brain ZeroOuUm venceu praticamente em quase todos os cenários, com exceção do resultado no primeiro cenário.

Tabela C.2: Resultado da execução do T1000 Max Recompensa Acumulada nos cenários estáticos: do cenário 1 ao 5.

Resultado	**1**	**2**	**3**	**4**	**5**	**Total**
Empate	27 (2,7%)	35 (3,5%)	21 (2,1%)	42 (4,2%)	34 (3,4%)	159 (3,18%)
FB_5	66 (6,6%)	5 (0,5%)	36 (3,6%)	0 (0%)	36 (3,6%)	143 (2,86%)
H_2	229 (22,9%)	171 (17,1%)	229 (22,9%)	226 (22,6%)	171 (17,1%)	1026 (20,52%)
I_4	**349 (34,9%)**	357 (35,7%)	330 (33%)	316 (31,6%)	356 (35,6%)	1708 (34,16%)
Z_1	329 (32,9%)	**432 (43,2%)**	**384 (38,4%)**	**416 (41,6%)**	**403 (40,3%)**	**1964 (39,28%)**
Total	1000 (100%)	1000 (100%)	1000 (100%)	1000 (100%)	1000 (100%)	5000 (100%)

Os resultados das batalhas do **T1000 Max Recompensa Acumulada** nos cenários aleatórios, que são mostrados na Tabela C.3, foram semelhantes ao da tabela anterior. O desempenho do Brain Z_1 foi um pouco melhor porque desta vez ele conseguiu vencer em todos os cenários. O desempenho do I_4 diminuiu um pouco e ficou próximo do número de vitórias obtido pelo Brain H_2 que teve uma melhora no seu desempenho.

Tabela C.3: Resultado da execução do T1000 Max Recompensa Acumulada nos cenário aleatórios: de 6 ao 10.

Resultado	**6**	**7**	**8**	**9**	**10**	**Total**
Empate	36 (3,6%)	41 (4,1%)	36 (3,6%)	36 (3,6%)	35 (3,5%)	184 (3,68%)
FB_5	16 (1,6%)	13 (1,3%)	11 (1,1%)	12 (1,2%)	10 (1,0%)	62 (1,24%)
H_2	276 (27,6%)	231 (23,1%)	254 (25,4%)	241 (24,1%)	252 (25,2%)	1254 (25,08%)
I_4	272 (27,2%)	272 (27,2%)	276 (27,6%)	284 (28,4%)	279 (27,9%)	1383 (27,66%)
Z_1	**400 (40%)**	**443 (44,3%)**	**423 (42,3%)**	**427 (427%)**	**424 (42,4%)**	**2117 (42,34%)**
Total	1000 (100%)	1000 (100%)	1000 (100%)	1000 (100%)	1000 (100%)	5000 (100%)

Considerando tanto os cenários estáticos quanto aleatórios, na Tabela C.4 são apresentados a soma total de empates e de vitórias de cada Brain. Esse resultado final nos informa que o modelo treinado pelo Brain com representação ZeroOuUm que teve a melhor recompensa acumulada na fase de treinamento entre os outros 5 modelos gerados pelos treinamentos do Brain ZeroOuUm, é o Agente/Brain mais vitorioso entre os participantes deste **T1000 Max Recompensa Acumulada**. Note que o desempenho do modelo treinado com representação ICAART obteve a segunda colocação. Além disso, o segundo modelo com representação Híbrida obteve um desempenho superior ao último colocado. Ademais, no **T1000 Vencedores** (Ver Seção 5.2), note que o desempenho dos Brains Z_1 $(31,94\%)$ e do I_4 $(23,21\%)$ não foram tão bons quanto o desempenho deles obtidos no torneio **T1000 Max Recompensa Acumulada** (ver Tabela C.4, $Z_1 = 40,81\%$ das vitórias e $I_4 = 30,81\%$ das vitórias). Essa queda do desempenho do ZeroOuUm e do ICAART no **T1000 Vencedores** se deve ao fato de que o desempenho do Brain H_1 $(44,42\%)$ aumentou em comparação ao resultado do Brain H_2 $(22,8\%)$ no **T1000 Max Recompensa Acumulada**. Também é possível observar que o desempenho do Brain de Flag Binária Normalizada no **T1000 Vencedores** foi superior ao número de empates enquanto o Brain de Flag Binária no **T1000 Max Recompensa Acumulada** foi inferior ao número de empates. Também pode ser observado que os Brains que obtiveram as melhores estimativas de valor durante o treinamento foram, na verdade, os três piores Brains que participaram dos torneios T1000 acima mencionados (FB, FBN e ICAART).

Tabela C.4: Resultado T1000 Max Recompensa Acumulada. Total e porcentagem de Vitórias considerando todos cenários

Resultado	**Número de Vitórias**
Empate	343 (3,43%)
FB_5	205 (2,05%)
H_2	2280 (22,8%)
I_4	3091 (30,91%)
$\boldsymbol{Z_1}$	**4081 (40,81%)**
Total	10000 (100%)

Podemos observar que o experimento não ingênuo (realizado na Seção 5.2) parece ser mais justo e os resultados obtidos nele foram diferentes dos obtidos neste experimento injusto. Naquele, foi obtido um modelo que venceu até mesmo o vencedor do **T1000 Max Recompensa Acumulada** (O H_1 campeão do outro experimento venceu o Z_1 que foi o vencedor deste experimento, que inclusive participou do outro experimento por ter vencido o seu torneio prévio **T100 ZeroOuUm**).

C.2 Treinamento Multi-Brain

Nesta seção, mostramos os gráficos de cada uma das estatísticas que não foram apresentados no Capítulo 5, Seção 5.3.

No gráfico da Figura C.8, foi observado que o comportamento de todos os Brains foi muito semelhante ao comportamento de seu respectivo Brain no gráfico da Figura C.1, exceto o Brain de Flag Binária Normalizada que terminou o treinamento com a entropia próxima de 0,25 e durante quase todo o treinamento destoou bastante dos demais.

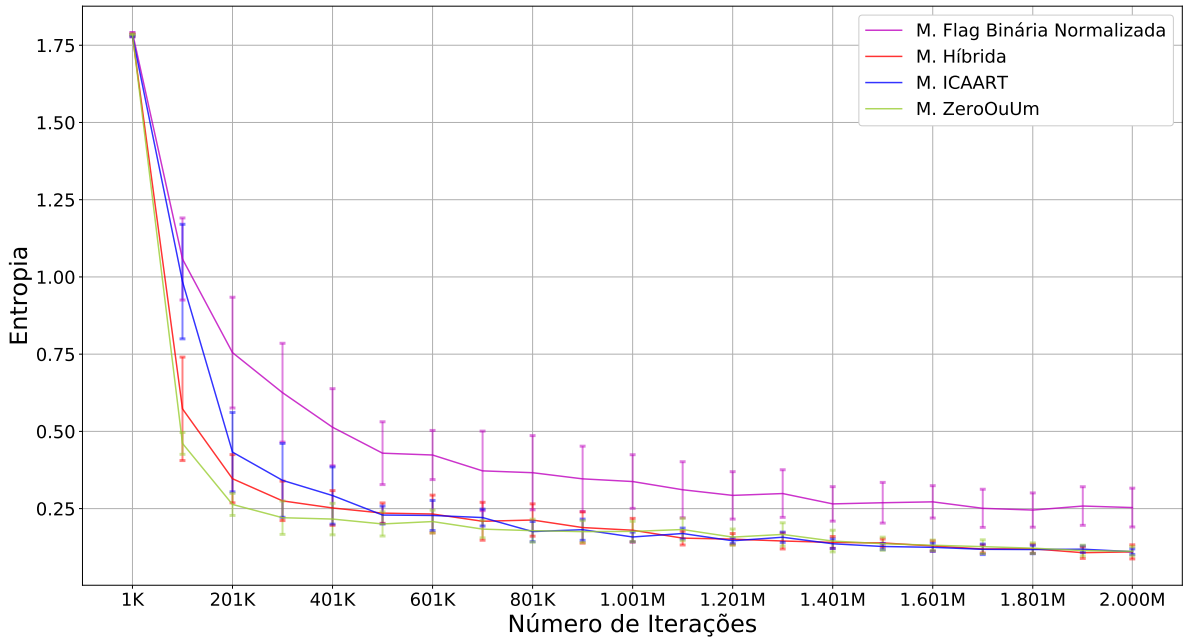

Figura C.8: Gráfico de Entropia obtido nos treinamentos MultiBrain no ambiente Bomberman.

No gráfico da Figura C.9, relativo ao tamanho dos episódios, ao se considerar o desvio padrão, houve um empate na primeira posição entre os Brains Híbrido e ZeroOuUm, e houve um empate na terceira posição entre os Brains ICAART e de Flag Binária Normalizada. Os únicos Brains que tiveram uma tendência de subida até o final do treinamento foram os ZeroOuUm e Híbrido. No início do treinamento o ICAART durou mais tempo na batalha, talvez por ter ficado isolado em seu canto, porém no final do treinamento seu resultado médio foi inferior aos dois que tiveram uma tendência de subida. Já o de Flag Binária Normalizada teve uma tendência de descida junto com o ICAART e no final do treinamento o tamanho médio de seu episódio foi de 50 iterações (geralmente o primeiro agente a morrer).

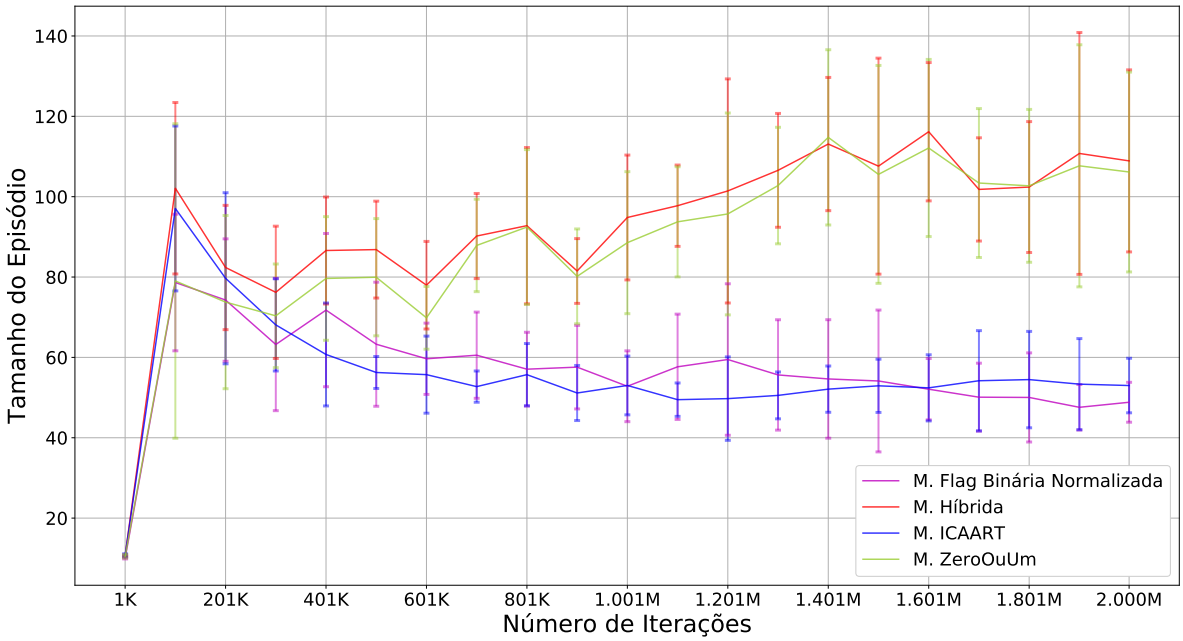

Figura C.9: Gráfico do Tamanho dos Episódios obtido nos treinamentos MultiBrain no ambiente Bomberman.

No gráfico da Figura C.10, os dois Brains que obtiveram uma estimativa de valor adequada (tendência de subida) no início do treinamento foram os Híbrido e ZeroOuUm. O Brain Híbrido teve uma tendência de subida até a iteração 901K e depois disso, uma descida até a iteração 1,401M mas voltou a subir e se estabilizar até o término do treinamento. Já o Brain ZeroOuUm, após a iteração 401K, teve apenas tendência de descida até se estabilizar na iteração 1,701M. Os outros dois Brains tiveram uma tendência de descida no início do treinamento até a iteração 201K e depois tiveram uma tendência mínima de subida até o final do treinamento.

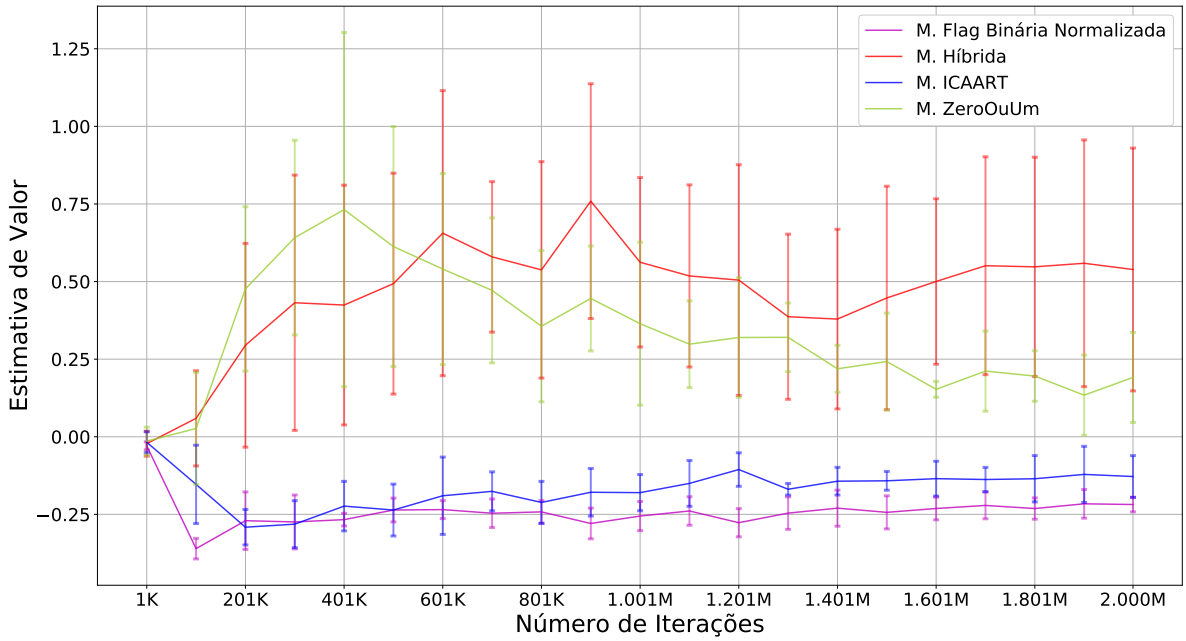

Figura C.10: Gráfico de Estimativa de Valor obtido nos treinamentos Multibrain no ambiente Bomberman.

No gráfico da Figura C.11, os Brains que não tiveram um loss de valor adequado foram o ICAART e o de Flag Binária Normalizada, porque a recompensa acumulada deles nunca parou de crescer (em quantidades pequenas) e as suas funções de loss de valor praticamente apenas foram diminuindo ou se estabilizaram com o passar do tempo, sem obter grandes picos como os dos Brains Híbrido e ZeroOuUm. Outra observação que pode ser feita é que os Brains Híbrido e ZeroOuUm tiveram loss de valor maiores que no treinamento individual. Estão em média próximos de 0,125 enquanto que no gráfico da Figura C.6 os seus valores nunca alcançaram o loss de valor 0,125.

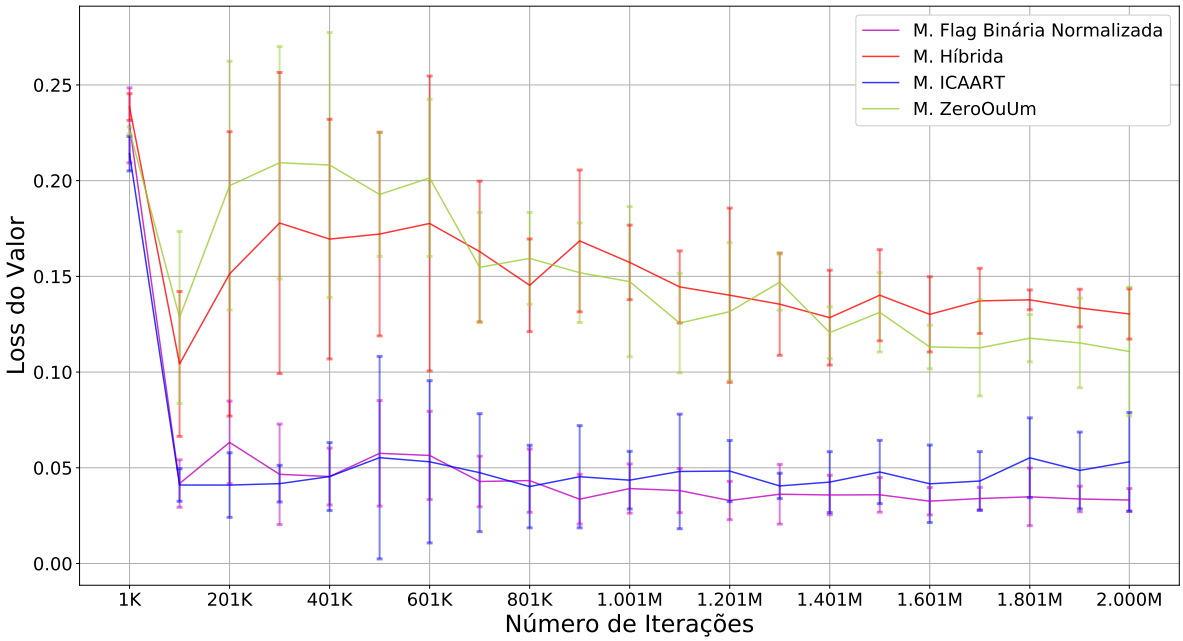

Figura C.11: Gráfico de Loss de Valor sem zoom obtido nos treinamentos MultiBrain no ambiente Bomberman.

No gráfico da Figura C.12 foi observado também que a magnitude do Loss de Política de todos os Brains possuem uma tendência de queda e um comportamento esperado (mesmo que seja mínimo).

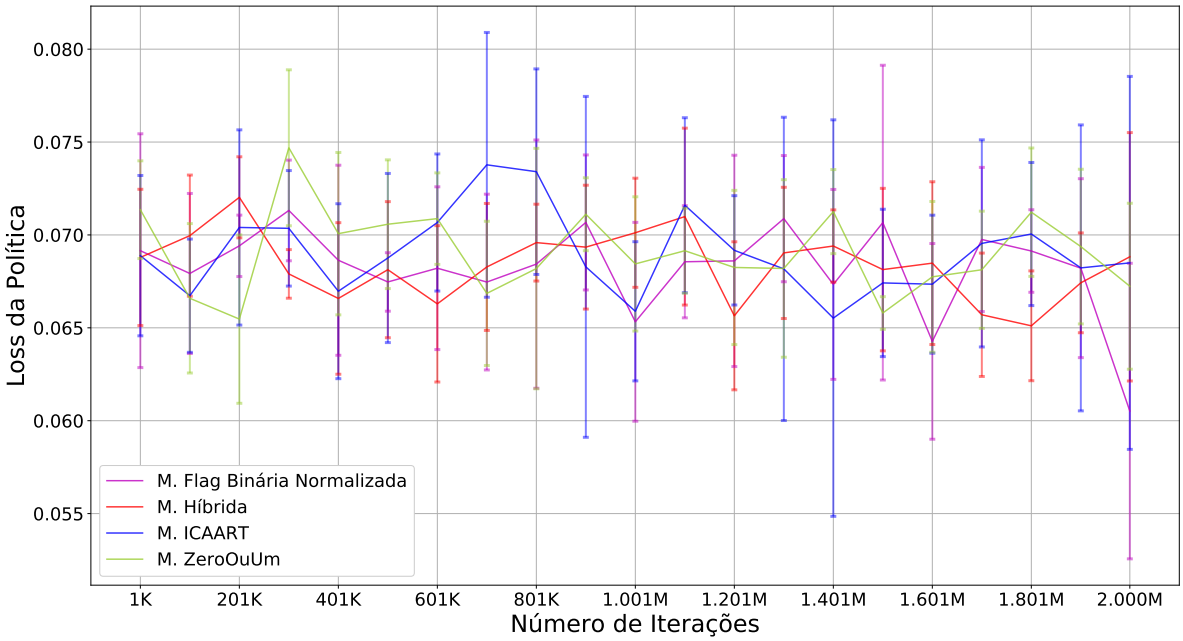

Figura C.12: Gráfico de Loss de Política obtido nos treinamentos Multi-Brain no ambiente Bomberman.

C.3 Treinamento Um-Brain com PPO e LSTM

Nesta seção, mostramos os gráficos de cada uma das estatísticas que não foram apresentados no Capítulo 5, Seção 5.4.

Os gráficos do LSTM foram plotados junto com as medidas do treinamento de representação Híbrida, que foi realizado no primeiro experimento no ambiente Bomberman, para efeito de comparação. Como pode ser observado na maioria das estatísticas, os resultados são semelhantes, com exceção do gráfico de Loss de Política, em que a magnitude obtida pelo LSTM foi maior que os valores da representação Híbrida.

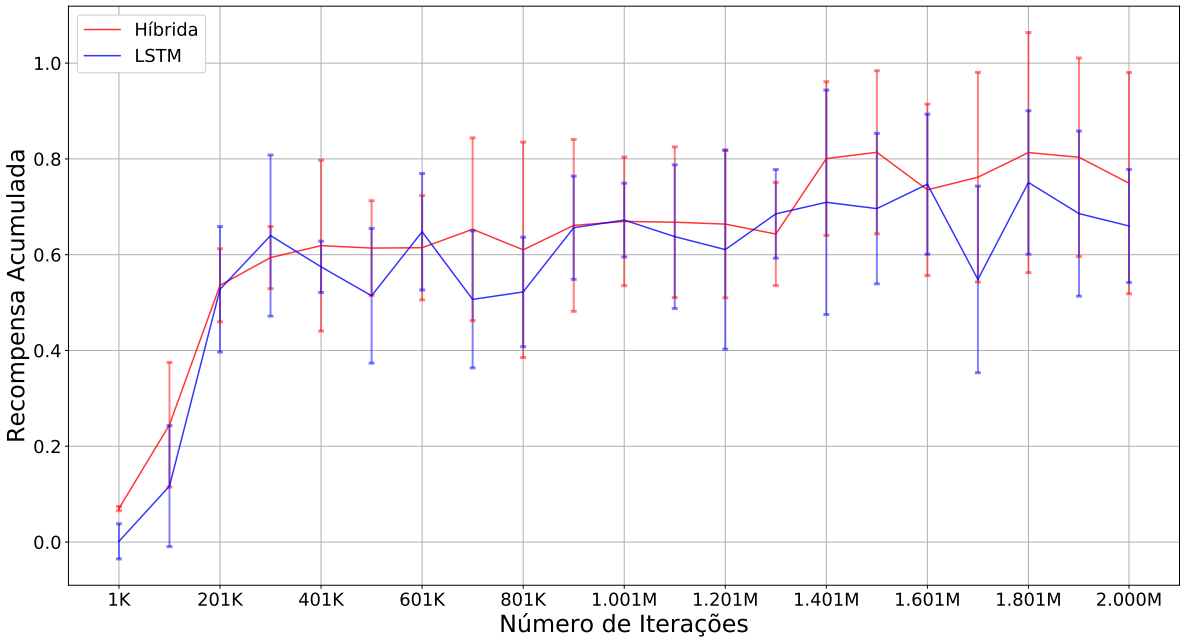

Figura C.13: Gráfico de Recompensa Acumulada obtido nos treinamentos LSTM no ambiente Bomberman.

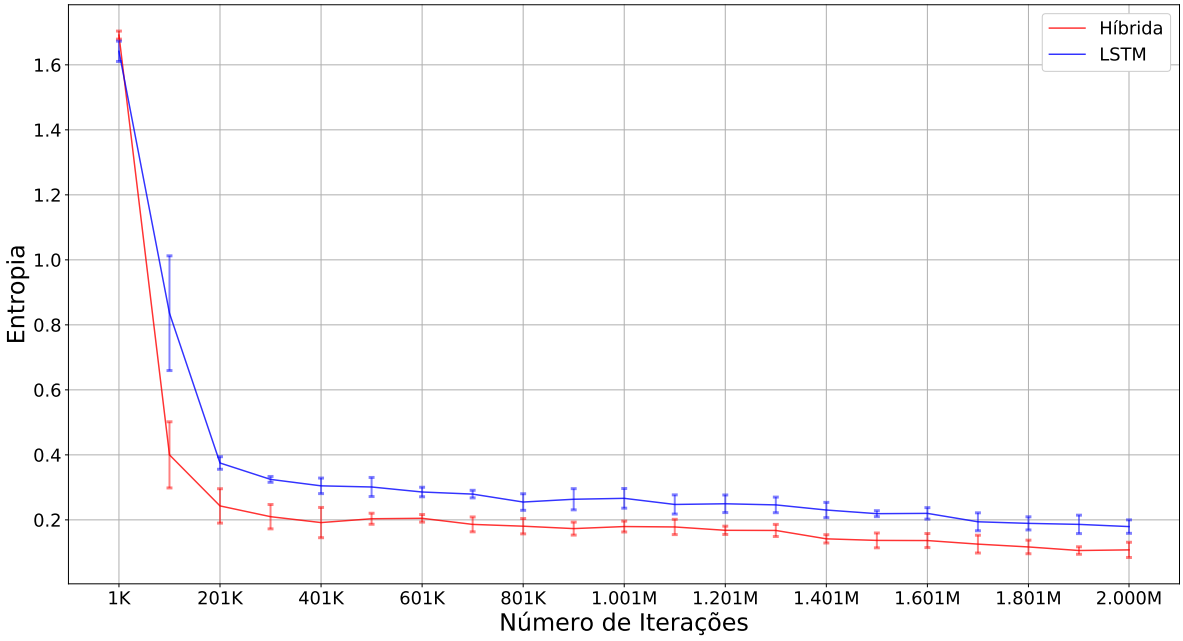

Figura C.14: Gráfico de Entropia obtido nos treinamentos LSTM no ambiente Bomberman.

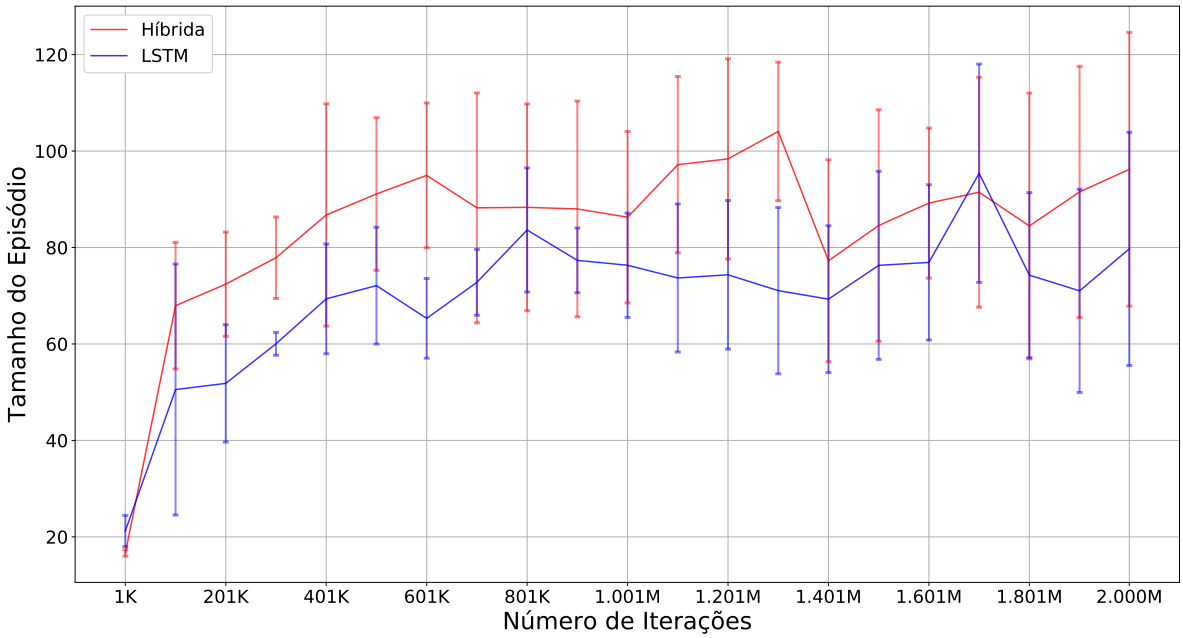

Figura C.15: Gráfico do Tamanho do Episódio obtido nos treinamentos LSTM no ambiente Bomberman.

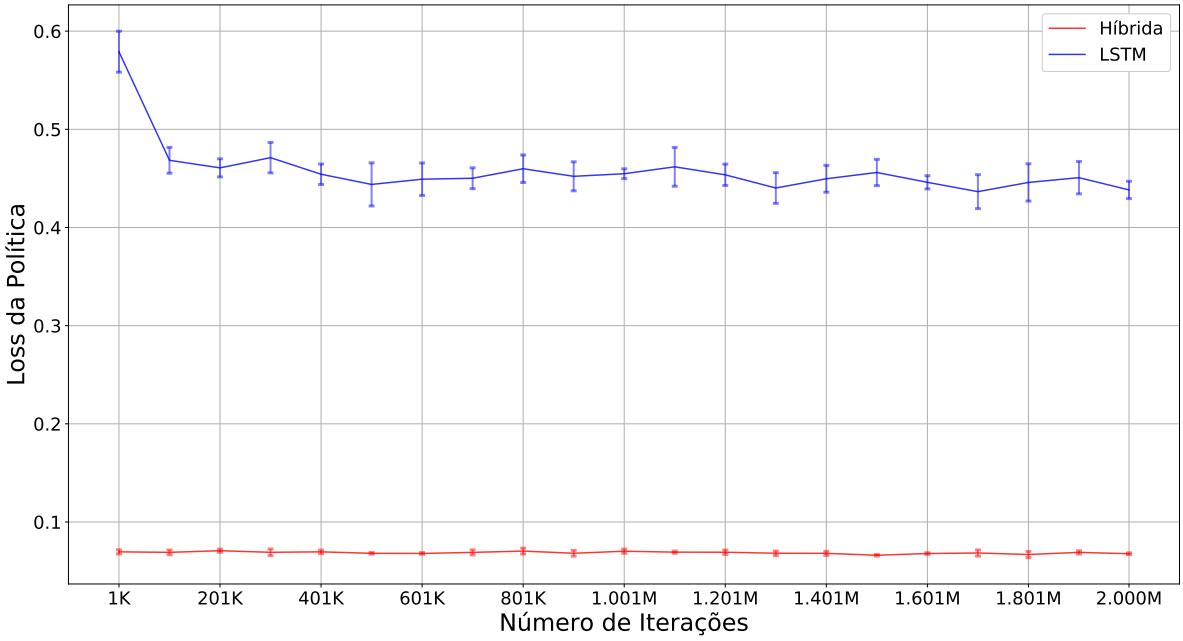

Figura C.16: Gráfico de Loss de Política obtido nos treinamentos LSTM no ambiente Bomberman.

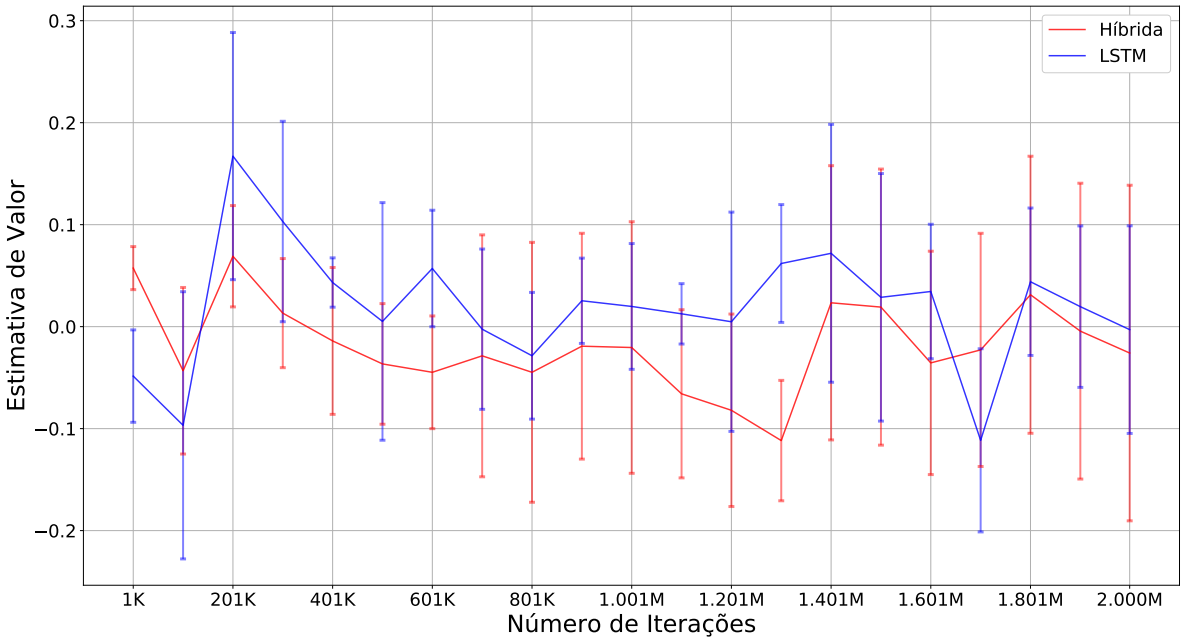

Figura C.17: Gráfico de Estimativa de Valor obtido nos treinamentos LSTM no ambiente Bomberman.

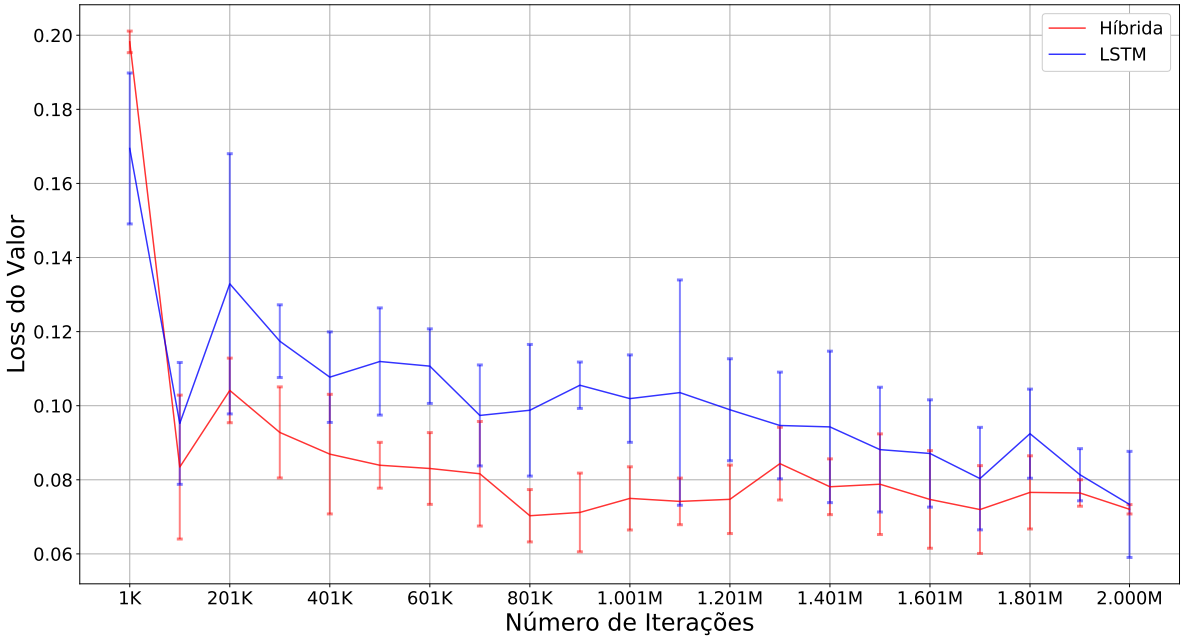

Figura C.18: Gráfico de Loss de Valor obtido nos treinamentos LSTM no ambiente Bomberman.

ANEXO A – Um Resumo do ML-Agents Toolkit

A.1 Introdução

Neste anexo fizemos um resumo, em tradução livre, do artigo [18] sobre o ML-Agents Toolkit contendo as principais informações.

O Unity Machine Learning Agents Toolkit é um projeto de código aberto que permite que pesquisadores e desenvolvedores criem ambientes de simulação usando o Editor da Unity e interajam com eles lidando com uma API Python. O ML-Agents foi criado para tirar proveito de todas as vantagens oferecidas pelo motor de jogo Unity. Além disso, traz muitos exemplos de uso que podem ser aproveitados como ambientes de benchmarking para algoritmos de RL ou como ponto de partida para a criação de novos ambientes de aprendizado. Também traz algoritmos de base para RL e IL que podem ser utilizados para treinar agentes dentro de qualquer ambiente criado com o núcleo do ML-Agents, e esses algoritmos servem como ponto de partida para aqueles que querem desenvolver novos algoritmos.

O ML-Agents é composto por duas partes principais: ML-Agents SDK e um pacote Python. Com o SDK, podemos criar ambientes de aprendizado dentro do editor do Unity e desenvolver scripts na linguagem C#. E o pacote Python permite que os algoritmos de ML desenvolvidos em Python se comuniquem com o SDK, e vice-versa.

A.2 ML-Agents SDK

Quando importamos o ML-Agents SDK para dentro do Unity, transformamos o Scene num ambiente de aprendizado. Essa transformação é feita utilizando-se três tipos de entidades do SDK: Agents (agentes), Brains (cérebros) e Academy (academia).

Usamos diretamente o componente Agent para indicar que um GameObject dentro de um Scene é um agente. Este componente é responsável por coletar informações do

ambiente e por atuar neste por meio de ações. Cada Agent é conectado diretamente a um único Brain, que é responsável por tomar as decisões para todos agentes conectados a ele. Ou seja, é o Brain que entrega a política a ser seguida para os Agents conectados a ele. Uma tomada de decisão de um Brain pode ser realizada de várias formas por meio de: algum dispositivo de entrada de um jogador, scripts predefinidos, modelos de redes neurais incorporados internamente, ou através de interação da API Python. Esses tipos de Brain são representados pelo ML-Agents SDK como Player, Heuristic, Internal e External respectivamente.

Um ambiente de aprendizado pode ter vários Brains, assim cada Brain pode aprender uma política diferente, e a cada Brain podem estar conectados um ou vários Agents. Todavia, apenas um Academy pode existir no Scene. O componente Academy dentro de um Scene é usado para acompanhar as etapas da simulação e para fornecer funcionalidades como a capacidade de redefinir o ambiente, definir a velocidade desejada de simulação e a taxa de quadros. O Academy também contém a capacidade de definir parâmetros de reinicialização, que podem ser usados para alterar a configuração do ambiente durante o tempo de execução.

Uma vez configurado os GameObjects e seus componentes dentro do Scene, é possível definir o ambiente de aprendizado e criar Processos de Decisão de Markov (MDP) ou MDPs Parcialmente Observáveis (POMDP) que servirão de base para uma variedade de possíveis tarefas de RL como as explicadas no livro de Sutton [51]. Para resolver esses problemas de Markov dentro do ML-Agents é necessário informar para o agente as observações, as ações e as recompensas dentro do componente Agent dele. Além disso, recompensas podem ser ativadas em qualquer script que tenha acesso ao componente Agent do agente em questão assim como o estado terminal dele pode ser alterado.

As observações fornecem ao agente um contexto sobre o estado do ambiente. Elas podem assumir a forma de um vetor de números de ponto flutuante, bem como qualquer número de saídas de câmera renderizadas de dentro da cena em que o agente contém uma referência para ela. As ações podem assumir a forma de variáveis discretas (vetor de inteiros) ou contínuas (vetor de ponto flutuante).

É possível que os agentes solicitem decisões de Brains em um intervalo fixo ou dinâmico, conforme definido pelo desenvolvedor do ambiente. A função de recompensa, usada para fornecer um sinal de aprendizado ao agente, pode ser definida ou modificada a qualquer momento durante a simulação usando o sistema de script Unity.

A.3 Pacote Python

O pacote Python fornecido contém uma classe chamada UnityEnvironment que pode ser usada para iniciar e fazer interface com executáveis da Unity (assim como com o Editor). A comunicação entre o Python e a Unity ocorre por meio de um protocolo de comunicação gRPC e utiliza mensagens protobuf. A noção básica para se interagir com a API Python é descrita abaixo:

- env = UnityEnvironment(filename) - Inicia um ambiente de aprendizado com um nome de arquivo ou caminho e estabelece comunicação com a API python.
- env.reset() - Redefine o ambiente e retorna um dicionário de objetos BrainInfo contendo observações iniciais de todos os agentes
- env.step(actions) - Executa um passo (step) no ambiente de aprendizagem fornecendo um conjunto de ações para todos os agentes presentes no dicionário BrainInfo recebido anteriormente e retorna um dicionário de objetos BrainInfo contendo novas observações dos agentes que requerem decisões.
- env.close() - Envia o sinal de término para o ambiente de aprendizado.

BrainInfo é uma classe que contém listas de observações, ações anteriores, recompensas e diversas variáveis de estado. Em cada etapa da simulação, o BrainInfo contém informações para todos os agentes ativos dentro da cena que requerem decisões para que a simulação prossiga.

A.4 Algoritmos de base

Os desenvolvedores do ML-Agents Toolkit implementaram um conjunto de algoritmos de base que podem ser utilizados para validar ambientes ou como ponto de partida para se desenvolver novos algoritmos. São disponibilizados dois algoritmos de base. Um para RL e outro para IL. Para RL há uma implementação do Proximal Policy Optmization (PPO) [45], um algoritmo do estado da arte de DRL com uma opção para estendê-lo utilizando Intrinsic Curiosity Module (ICM) [38], e/ou utilizando um Long-Short-Term Cell (LSTM) [14]. Além disso eles forneceram uma implementação do Behavioral Cloning (BC), um algoritmo simples de aprendizagem de imitação [16].