PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
ESCOLA POLITÉCNICA
BACHARELADO EM ENGENHARIA DE COMPUTAÇÃO

APRENDIZADO POR REFORÇO EM JOGOS DE ESTRATÉGIA

TIAGO RECK GAMBIM

Trabalho de Conclusão II apresentado como requisito parcial à obtenção do grau de Bacharel em Engenharia de Computação na Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador: Prof. Felipe Rech Meneguzzi

Porto Alegre
2021

DEDICATÓRIA

Dedico este trabalho a meus pais.

“The art of simplicity is a puzzle of complexity.”
(Douglas Horton)

AGRADECIMENTOS

Agradeço aos meu pais por todo o apoio que me deram durante toda minha vida acadêmica. Agradeço também ao professor orientador Felipe Rech Meneguzzi por todo o auxílio no desenvolvimento deste trabalho, que foi fundamental para o resultado obtido. Por fim, agradeço aos amigos e colegas Eduardo Brum, Eric Friedrich, Marcelo Holgado, Nicolas Moura, Rafael Basso e Vinicius Rocha por todo o suporte durante a graduação.

APRENDIZADO POR REFORÇO EM JOGOS DE ESTRATÉGIA

RESUMO

Jogos de estratégia são um tema muito explorado em pesquisas na área de inteligência artificial, seja por sua complexidade, variedade ou pelo aspecto competitivo. As abordagens adotadas variam de algoritmos de IA clássica, como busca e planejamento, a algoritmos de aprendizado, como aprendizado por reforço e por reforço profundo.

Neste trabalho foi desenvolvido um agente de inteligência artificial utilizando aprendizado por reforço para o jogo de estratégia *Battle for Wesnoth*. A abordagem proposta é utilização do agente na execução ações de baixo nível, como movimentação e combate, tratando cada unidade individualmente. Os resultados foram comparados com as IAs disponibilizadas no próprio jogo.

Palavras-Chave: Battle for Wesnoth, aprendizado, aprendizado por reforço.

REINFORCEMENT LEARNING IN STRATEGY GAMES

ABSTRACT

Strategy games are a widely explored artificial intelligence research field, either due to their complexity, variety, or competitive aspect. The taken approaches diversify from classical AI algorithms, such as search and planning, to learning algorithms, such as reinforcement learning and deep reinforcement learning.

In this work was developed an AI agent with reinforcement learning to the strategy game Battle for Wesnoth. The proposed idea is to use the agent to control low-level actions, such as move and fight, dealing with each unit individually. Results were compared with Battle for Wesnoth's own artificial intelligence.

Keywords: Battle for Wesnoth, learning, reinforcement learning.

LISTA DE FIGURAS

Figura 2.1 – Exemplo de cenário do jogo *Battle for Wesnoth*. 14
Figura 2.2 – Neurônio Artificial 23
Figura 2.3 – Rede Neural Artificial 24
Figura 3.1 – Diagrama de funcionamento da integração com o *Battle for Wesnoth*. 29
Figura 3.2 – Diagrama de UML da implementação do ambiente em *Python*. 30
Figura 3.3 – Diagrama de UML da *Candidate Action* implementada. 33
Figura 3.4 – Diagrama de rede neural na versão linear. 39
Figura 3.5 – Diagrama de rede neural na versão não-linear. 39
Figura 4.1 – Média dos resultados dos cinco treinamentos da versão linear. 42
Figura 4.2 – Melhor resultado dos cinco treinamentos da versão linear. 42
Figura 4.3 – Pior resultado dos cinco treinamentos da versão linear. 42
Figura 4.4 – Média dos testes da rede linear com melhor resultado. 42
Figura 4.5 – Média dos testes da rede linear com pior resultado. 42
Figura 4.6 – Média dos resultados dos cinco treinamentos da versão não-linear. . 44
Figura 4.7 – Melhor resultado dos cinco treinamentos da versão não-linear. 44
Figura 4.8 – Pior resultado dos cinco treinamentos da versão não-linear. 44
Figura 4.9 – Média dos testes da rede com melhor resultado. 45
Figura 4.10 – Média dos testes da rede com pior resultado. 45

LISTA DE TABELAS

Tabela 4.1 – Taxa de Vitórias das Implementações 45

LISTA DE SIGLAS

BFW – Battle for Wesnoth

CA – Candidate Action

DQN – Deep Q-Networks

IA – Inteligência Artificial

MOBA – Multiplayer Online Battle Arena

PDM – Processo de Decisão de Markov

RELU – Rectified Linear Units

RTS – Real-Time Strategy

SUMÁRIO

1 INTRODUÇÃO **12**

2 REVISÃO BIBLIOGRÁFICA **14**

2.1 BATTLE FOR WESNOTH 14

2.2 INTELIGÊNCIA ARTIFICIAL 16

2.2.1 AMBIENTES 16

2.2.2 APRENDIZADO DE MÁQUINA 18

2.2.3 APRENDIZADO POR REFORÇO 18

2.3 PROCESSO DE DECISÃO DE MARKOV 20

2.4 APROXIMAÇÃO DE FUNÇÕES 21

2.5 REDES NEURAIS ARTIFICIAIS 22

2.5.1 REDES NEURAIS PROFUNDAS 25

2.5.2 REDES NEURAIS CONVOLUCIONAIS 25

2.6 Q-LEARNING E DEEP Q-LEARNING 26

2.7 OPENAI GYM 27

3 DESENVOLVIMENTO **29**

3.1 AMBIENTE OPENAI GYM 29

3.1.1 AMBIENTE EM PYTHON 30

3.1.2 ADD-ON 32

3.2 AGENTE DE APRENDIZADO POR REFORÇO 35

3.2.1 IMPLEMENTAÇÃO TABULAR 35

3.2.2 IMPLEMENTAÇÃO COM APROXIMAÇÃO DE FUNÇÃO 37

4 RESULTADOS **40**

4.1 VERSÃO TABULAR 41

4.2 VERSÃO LINEAR 41

4.3 VERSÃO NÃO-LINEAR 43

4.4 COMPARAÇÃO 45

5 TRABALHOS RELACIONADOS **46**

6 CONCLUSÃO **48**

REFERÊNCIAS **49**

1. INTRODUÇÃO

Jogos têm mostrado grande potencial como ambiente de testes para algoritmos de inteligência artificial e avanços têm sido feitos nos últimos anos. Eles apresentam uma série de problemas desafiadores, provém ambientes que modelam aspectos simplificados do mundo real e podem ser utilizados para aprender a resolver seus problemas [RP20]. Por fim, podem ser explorados no aspecto competitivo: IAs conseguiram derrotar alguns dos melhores jogadores, por exemplo, o agente *Deep Blue* que derrotou Garry Kasparov em uma partida de xadrez [CHhH02], o agente *OpenAI Five* que se saiu vitorioso contra uma equipe campeã mundial de Dota 2, *Team OG* [BBC+19] e a IA *AlphaStar* que conseguiu derrotar um jogador profissional no jogo *Starcraft II* [ACT19].

Jogos de estratégia compõem um importante subgênero de jogos e possuem algumas particularidades que os tornam interessantes do ponto de vista de IA. A primeira, e talvez a maior delas, é a quantidade de possibilidades a serem consideradas: mundos enormes e complexos, um grande número de decisões a serem tomadas e objetivos a serem cumpridos pelo jogador. Em jogos como *Civilization* e *Age of Empires* têm-se imensos mapas nos quais os jogadores podem construir, explorar, recrutar tropas e trabalhadores, coletar recursos, realizar comércio, efetuar ataques, pesquisar tecnologias, dentre diversas outras ações. Alguns jogos possuem, ainda, diversos modos de se obter a vitória e outros, por se tratarem de jogos multijogador, podem possuir conceitos de aliados e inimigos. Considerando estes fatores, implementar uma inteligência artificial pode apresentar grandes desafios.

Um exemplo interessante do gênero, que será utilizado como objeto de estudo neste trabalho, é *Battle for Wesnoth* (BfW), um jogo de estratégia baseado em turnos. Ele possui aspectos em comum com os jogos anteriormente citados, *Civilization* e *Age of Empires*, como gerenciamento de recursos e tropas, controle de território e batalhas entre unidades. Por se tratar de um jogo em turnos, desenvolver uma IA capaz de jogá-lo torna-se uma tarefa mais fácil: como a decisão de ir para o próximo turno depende do jogador, o tempo necessário para tomar a decisão não impacta no resultado, sendo então um aspecto a menos a ser considerado no desenvolvimento. Além disso, BfW possui código aberto, permitindo assim a análise de seu código-fonte, extração de dados e envio de comandos para ele. Assim, torna-se mais fácil o desenvolvimento de uma interface para ser utilizada pela IA para controlar o jogo.

A abordagem implementada neste trabalho consiste no desenvolvimento de uma interface para extrair observações sobre o ambiente do BfW e enviar ações em resposta. A partir dela foi realizada uma integração com a ferramenta *OpenAi Gym*, para gerar um ambiente com uma interface padronizada, facilitando o restante do processo de desenvolvimento. Por fim, foi implementado um agente de aprendizado por reforço utilizando o

ambiente criado no *Gym*. Foram implementadas três versões do agente: a primeira delas utilizando uma metodologia tabular e as duas últimas utilizando aproximação de funções linear e não linear, respectivamente. Para cada versão do agente foram realizados diversos treinamentos, testes e, a partir dos resultados, foi realizada uma comparação entre as três implementações.

Este trabalho está organizado da seguinte forma. No Capítulo 2 apresenta-se uma breve revisão bibliográfica. O Capítulo 3 especifica detalhadamente o processo de desenvolvimento aplicado neste trabalho. No Capítulo 4 são apresentados os resultados obtidos nos experimentos com a implementação proposta. Por fim, no Capítulo 5 abordam-se alguns trabalhos relacionados e no Capítulo 6 apresenta-se a conclusão do trabalho.

2. REVISÃO BIBLIOGRÁFICA

A seguir serão abordados conceitos importantes que serão utilizados no restante deste trabalho. Primeiramente serão explicados detalhes sobre o jogo *Battle for Wesnoth*, seguido por uma revisão sobre diversos conceitos de inteligência artificial. Por fim, será apresentada a ferramenta OpenAi Gym.

2.1 Battle for Wesnoth

Battle for Wesnoth, como mencionado no Capítulo 1, é um jogo de estratégia em turnos de código aberto. Ele se passa em um cenário medieval de alta fantasia, é jogado em mapas hexagonais e baseia-se em gerenciamento de tropas e recursos. Nos mapas são encontrados diferentes categorias de terrenos como rios, florestas, planícies e montanhas, além de construções como aldeias e castelos.

Jogadores possuem uma unidade líder com a qual é possível recrutar novas tropas. O recrutamento é sempre efetuado pela unidade líder em uma torre de menagem, posicionando as novas unidades em castelos adjacentes. Cada unidade possui uma facção, como orques ou elfos, um nível, pontos de experiência e características únicas baseadas em seu tipo e facção, como ataques, habilidades e bônus de terreno. As ações básicas de cada unidade são mover-se e atacar unidades em seu alcance. Cada jogador possui ouro, sendo um recurso gerado todo turno baseado no número de aldeias sob seu controle. Ouro é utilizado para recrutar novas unidades e cada unidade gera uma despesa por turno para mantê-la. O jogo se encerra ao derrotar a unidade líder inimiga ou ao atingir um limite de turnos.

Figura 2.1 – Exemplo de cenário do jogo *Battle for Wesnoth*.

Dois aspectos fundamentais do jogo são o controle de terreno e de condições das batalhas. As unidades possuem bônus baseados em terreno, horário, unidades próximas, dentre outros fatores. Orques, por exemplo, são fortificados à noite, enquanto elfos recebem bônus em florestas. Existem alguns terrenos que são intransponíveis e outros que custam o dobro de movimentação, fazendo com que a rota mais curta nem sempre seja a melhor e até mesmo a mais rápida. Assim, efetuar um combate em terreno favorável pode definir o rumo da batalha: por exemplo, em um terreno dividido por um rio profundo com uma pequena ponte, defender a ponte pode levar uma situação onde os inimigos têm que atravessar individualmente e serão atacados por múltiplos defensores.

Apesar de não possuir uma pontuação, alguns parâmetros podem ser utilizados para aproximar a força de cada jogador, como o número de aldeias conquistadas, o ouro e o número de tropas. Assim, esses dados podem ser utilizados para avaliar os resultados de uma ação executada, bem como influenciar na tomada de decisão de cada jogador.

Quanto à parte técnica, o jogo é desenvolvido na linguagem C++ e possui parâmetros que podem ser utilizados na execução para ativar o modo de depuração, ocultar a interface gráfica, remover o som, dentre outras funções. Alguns *scripts* utilitários são desenvolvidos na linguagem *Python*, como um editor de mapas e algumas ferramentas para configuração de servidor. Por fim, é possível criar *add-ons* utilizando a linguagem Lua. A criação de *add-ons* possui diversas funcionalidades, permitindo criação de campanhas, novas facções, mapas, além de modificar algumas mecânicas do jogo, incluindo as IAs.

Para modificação e criação de IAs, em específico, são utilizadas *Candidate Actions* (CAs). CAs são *scripts* em Lua que representam possíveis ações que uma IA pode tomar. Elas funcionam com dois blocos, avaliação e execução, sendo o primeiro responsável por calcular uma pontuação para a ação e o segundo responsável por sua execução. O comportamento de uma IA no *Battle for Wesnoth* consiste em, a cada turno, calcular a pontuação de todas suas CAs. A de pontuação mais alta, então, é executada, assim efetuando uma ação. Este ciclo de avaliação e execução é repetido até que todas as CAs retornem uma pontuação igual a zero, indicando que nenhuma ação extra pode ser realizada e finalizando o turno. Através de arquivos de configuração é possível criar e modificar IA apenas editando alguns parâmetros, adicionando a lista de novas CAs e, se desejado, removendo as já existentes. Assim é possível a criação de IAs customizáveis de maneira relativamente simples.

A abordagem implementada utilizou a criação de *add-ons*. Através deles, foi possível a criação de uma IA customizada que se comporta como uma interface com um *script* externo, enviando informações sobre o jogo e recebendo ações para serem executadas. Foi utilizada também a criação de mapas customizados para o treinamento e validação da IA criada.

2.2 Inteligência Artificial

Historicamente, definições de IA surgiram em torno de duas principais características: raciocínio e comportamento. Dessas características, surgem 4 grupos de definições: sistemas que pensam como humanos, que agem como humanos, que pensam racionalmente, que agem racionalmente. As duas primeiras definições comparam o desempenho dos algoritmos em relação ao comportamento humano, enquanto as duas últimas comparam à racionalidade [RN10, Cap. 1].

Para este trabalho, devido à necessidade de uma IA capaz de agir dada uma situação no jogo, o conceito mais relevante é o de agir racionalmente. Agir racionalmente, em geral, envolve pensar racionalmente e chegar à conclusão correta. Esse, porém, nem sempre é o caso: em algumas situações uma solução ideal pode não existir ou, ainda, pode não haver tempo suficiente para computá-la. Ainda assim, uma ação precisa ser tomada. Para tomar decisões nestas situações, uma abordagem possível é a implementação de um agente racional.

Um agente consegue fazer observações sobre um ambiente através de sensores e agir nele através de atuadores [RN10, Cap. 2]. Estes sensores e atuadores podem existir tanto ao nível de *hardware* quanto *software*: um agente pode observar um ambiente físico através de sensores de proximidade e atuar através de motores, por exemplo, assim como pode atuar sobre outros programas e/ou arquivos, lendo e escrevendo em memória ou através de pacotes na rede. Observações são as leituras dos sensores em um determinado instante de tempo, e podem ser organizadas em sequência para montar a percepção do agente temporalmente. As variáveis coletadas em observações são utilizadas para definir o estado do ambiente. Agentes pode ser definidos, basicamente, como ~~uma~~ funções que mapeiam estados em ações.

Agentes racionais devem mapear observações às melhores ações possíveis. Uma das maneiras de medir a racionalidade de uma decisão é analisar suas consequências, imediatas e a longo prazo. Essa análise pode ser conduzida de diversas maneiras: pode-se prever os resultados ao ter o conhecimento necessário sobre o ambiente e seu comportamento, ou então utilizar de dados obtidos de experiências passadas. Cada ação terá consequências, boas ou ruins, e o objetivo do agente é descobrir a sequência de ações que retornarão os melhores resultados.

2.2.1 Ambientes

Como mencionado em anteriormente, ambientes são o objeto de observação e ação de agentes. Eles são o mundo com o qual o agente deve interagir, situações que ele

deve enfrentar. A partir de um ambiente é possível extrair variáveis úteis para o problema e definir estados que o ambiente pode assumir. O conjunto de estados possíveis é denominado espaço de estados. Ao fazer observações de um ambiente é possível determinar o estado em que ele se encontra naquele determinado instante de tempo.

Cada problema dá a um ambiente propriedades específicas, aspectos únicos a serem analisados. Baseado em seu comportamento e características, eles podem ser classificados nos seguintes grupos [RN10, Cap. 2]:

1. Completamente Observáveis e Parcialmente Observáveis.
2. Agente Único e Multi-Agente.
3. Determinístico e Estocástico.
4. Episódico e Sequencial.
5. Discreto e Contínuo.
6. Conhecido vs Desconhecido.

No que se refere a sua observabilidade, é necessário analisar a capacidade do agente de extrair todos os dados necessários para sua tomada de decisão ao realizar uma única observação do ambiente. Caso seja possível, considera-se completamente observável e, caso contrário, parcialmente observável. Por exemplo, um agente ao fazer uma obervação pode ter uma limitação e só perceber um determinado raio ao seu redor. Ambientes parcialmente observáveis possuem um grau maior de dificuldade, pois os agentes precisam resolver internamente o problema dos dados ausentes, por exemplo, através de múltiplas observações.

Agente único e multi-agente, como o próprio nome indica, refere-se ao número de agentes atuando sobre o ambiente. O problema, porém é mais complexo que a definição: em casos multi-agente eles podem se relacionar de forma cooperativa ou competitiva. Pode haver, ainda, o fator de comunicação em caso de cooperação ou então a necessidade de evitar comportamento previsível em situações de competição.

Um ambiente é considerado determinístico se, dado seu estado e uma ação a ser tomada, é possível garantir qual será o próximo estado apenas com essas duas informações. Caso outros fatores do ambiente interfiram nessa mudança, como, por exemplo, uma possibilidade de a ação falhar ou ter outro resultado, ele é estocástico.

A divisão entre estático e dinâmico depende do comportamento do ambiente enquanto o agente computa a próxima ação. Se ele continua se modificando durante o processo de decisão do agente, é considerado dinâmico. Caso permaneça igual independente do tempo utilizado na escolha, é considerado estático. Há ainda casos onde o ambiente

permanece o mesmo, mas alguma forma de pontuação depende do tempo utilizado para cada decisão. Estes casos são considerados semi-dinâmicos.

Ambientes discretos e contínuos são definidos baseados em seu espaço de estados. Caso este seja finito, são considerados discretos. Já os que têm estados infinitos são considerados contínuos.

A classificação entre ambientes conhecidos e desconhecidos depende dos dados que o agente possui sobre o ambiente. Se um agente ao executar uma ação conhece o resultado ou as probabilidades dos possíveis resultados, ele é considerado conhecido. Caso contrário, se o agente precisar aprender a dinâmica do ambiente, este é dito desconhecido.

2.2.2 Aprendizado de Máquina

Dentro de IA existem diversas metodologias para o desenvolvimento de agentes: busca, planejamento, aprendizado, dentre outras. Devido à complexidade do ambiente e o tamanho do seu espaço de estados, a que será abordada e utilizada neste trabalho será aprendizado de máquina, mais especificamente aprendizado por reforço.

O conceito de aprendizado para um agente consiste em melhorar seu desempenho baseado em suas observações e ações sobre o ambiente [RN10, Cap. 1]. Com aprendizado é possível fazer com que um agente melhore ao longo do tempo a partir de suas experiências, sem necessariamente precisar de conhecimento prévio ou um modelo do ambiente.

Baseado na metologia utilizada para o aprendizado, é possível classificá-lo em três paradigmas: supervisionado, não supervisionado e por reforço. No primeiro o agente utiliza dados de experiências exemplo contendo uma relação de entradas e saídas esperadas, a partir das quais o agente estabelece uma função entre entrada e saída. Em aprendizado não supervisionado o agente aprende diretamente dos dados de entrada, tentando encontrar uma estrutura, grupos ou relações em grandes pacotes de dados. Já em aprendizado por reforço, o agente recebe recompensas, sejam positivas ou negativas, sobre as ações efetuadas e deve modificar seu comportamento para maximizar a recompensa acumulada.

2.2.3 Aprendizado por reforço

Aprendizado por reforço, como mencionado na Seção 2.2.2, aperfeiçoa o agente mapeando estados a ações para maximizar a recompensa acumulada recebida por elas. Para isso, ele deve explorar as ações possíveis, descobrindo suas recompensas e encontrando as sequências de decisões com a maior recompensa total no final [SB18, Cap. 1].

Recompensas, em aprendizado por reforço, representam o objetivo do problema: a função do agente é maximizar a recompensa total recebida durante a execução. Dependendo da ação, elas podem ser positivas ou negativas. Com isso, o agente acaba por reforçar comportamentos com bons resultados e desencorajando os ruins. Por fim, ações podem influenciar recompensas imediatas e também as futuras, em alguns casos até mesmo fornecendo recompensas negativas a curto prazo e grandes ao final.

Para balancear recompensas a curto e longo prazo existe a função-valor. Enquanto recompensas indicam resultado imediato de uma ação, a função-valor indica benefícios a longo prazo de um estado, considerando a sequência de estados prováveis a serem seguidos a partir dele [SB18, Cap. 1]. Assim, estados que tenham recompensas piores, mas são seguidos por estados que vão acumulando grandes recompensas, também são considerados na hora da tomada de decisão. Como o objetivo é maximizar a recompensa final acumulada e não a imediata, as ações são executadas tendo em vista a função-valor. Valores, porém, são muito mais difíceis de serem obtidos: enquanto recompensas são dadas pelo ambiente após a execução de uma ação, valores são estimados e atualizados através de sequências observações realizadas pelo agente ao longo do tempo e tendem a se tornarem precisos conforme o número de execuções cresce e mais estados e transições forem exploradas.

Ao começar a resolver um problema um agente pode não conhecer as recompensas de cada ação. Além disso, novas possibilidades podem surgir ao longo do tempo. Por conta disso, surge um fator a mais no problema: exploração. Para determinar recompensas de cada ação um agente deve explorá-las e, em casos estocásticos, isto deve ser efetuado diversas vezes. Surge então um dilema: o agente deve maximizar o retorno utilizando ações já conhecidas e, simultaneamente, explorar novas em busca de recompensas maiores. Essas duas abordagens devem ser balanceadas por um fator de exploração ϵ, podendo este ser fixo ou variável, para obter o melhor comportamento possível a longo prazo.

O comportamento do agente é definido por uma política. A política é basicamente uma relação entre estados do ambiente e quais ações tomar a partir deles. Políticas mudam conforme a descoberta de novas recompensas e atualizações na função-valor. Agentes de aprendizado por reforço que buscam melhorar a mesma política que utilizam para avaliação dos estados e ações durante o processo de aprendizado são denominados *on-policy*. Em contrapartida, agentes chamados *off-policy* utilizam duas políticas: a política alvo, que está sendo aprendida, e a política comportamental, usada para gerar o comportamento exploratório necessário durante o processo de aprendizado.

Por fim, algoritmos de aprendizado podem ou não possuir modelo. Um modelo deve descrever o comportamento do ambiente sendo usado para prever como ~~um~~ ele irá reagir a determinada ação. Nesse caso, o algoritmo é chamado *model-based*. Em contrapartida, existem as soluções *model-free*, que não utilizam modelos e aprendem o comportamento do ambiente fazendo experimentos e aprendendo com os resultados. Existem ainda

abordagens híbridas: aprendem o modelo através de experimentação e, posteriormente, utilizam-no com outras soluções, como planejamento.

2.3 Processo de Decisão de Markov

Problemas de aprendizado por reforço são formalmente representados através de processo de decisão de Markov. Processo de decisão de Markov, ou PDM, é uma formalização matemática para problemas sequenciais de tomada de decisões, onde cada ação possui recompensas imediatas e também afeta as próximas ações, implicando em recompensas futuras [SB18, Cap. 3].

PDM funciona através de uma sequência de interações entre agente e ambiente: em cada etapa, a partir de um estado do ambiente, um agente executa uma ação e, como resultado, recebe o novo estado e uma recompensa. Uma transição entre estados provocada por uma ação pode ser representada pela tupla $\langle s, a, r, s' \rangle$, sendo s o estado inicial, a a ação executada, s' o novo estado e r sua recompensa.Um problema de aprendizado em PDM é definido pela tupla $\langle S, A, P, R \rangle$ [AS10], sendo:

- S o conjunto dos possíveis estados que o ambiente pode assumir.
- A o conjunto das ações que podem ser executadas pelo agente.
- R a função recompensa. Tomar uma ação a em um estado s resulta em uma recompensa $R(s, a)$
- P é o conjunto das probabilidades de transições entre estados.

A probabilidade de transição a partir de um estado s, utilizando uma ação a, para um estado s' com uma recompensa r é formalizada pela Equação 2.1.

$$p(s', r|s, a) = Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\} \tag{2.1}$$

Uma propriedade fundamental para o processo é a propriedade de Markov. Segundo ela, todos os estados de um PDM devem conter informações sobre as interações entre agente e ambiente que sejam importantes para decisões futuras [SB18, Cap. 3]. Assim, dado um estado S_t, todas as informações necessárias para o cálculo de sua probabilidade de transição para um estado S_{t+1} estão contidas nele, não precisando de nenhuma informação sobre as transições anteriores ao estado S_t.

Em problemas de aprendizado por reforço o objetivo de um agente é maximizar a recompensa cumulativa recebida a longo prazo. Em PDM, este conceito é formalizado como retorno. Retorno, ou G_t, é a recompensa total descontada em um instante de tempo

t, matematicamente representada pela Equação 2.2. O parâmetro γ é denominado fator de desconto e assume valores $0 \leq \gamma \leq 1$. Sua função é determinar o valor de recompensas futuras no instante de tempo atual, podendo fazê-las terem pouca influência quando se aproxima de zero ou muita quando se aproxima de um. O fator de desconto, quando $\gamma < 1$, também é responsável por fazer o retorno convergir para um valor finito mesmo considerando uma sequencia de estados infinitos.

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.2}$$

Políticas, denotadas por π em PDM, são formalizadas como as probabilidades de tomar cada ação a partir de cada estado: $\pi(a|s)$ define a probabilidade de um agente tomar a ação a em um estado s seguindo uma política π. Uma política é considerada ótima quando ela é a política que possui o maior retorno esperado para todos os estados. Políticas ótimas são representadas por π_*.

A função-valor, por fim, determina o valor a longo prazo de um estado e, no caso do processo de decisão de Markov, determina o retorno esperado a partir dele. Formalmente, a função-valor de um estado s seguindo uma política π é definida pela equação de Bellman:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \tag{2.3}$$

Onde, $s, s' \in S$, $a \in A$, $r \in R$. A equação de Bellman representa o valor de um estado em função de sua recompensa imediata e do valor de seus estados sucessores, levando em conta as probabilidades de transições.

2.4 Aproximação de Funções

Em problemas simples de aprendizado por reforço, a função-valor pode ser encontrada utilizando uma matriz para representar os valores dos pares de estados e ações. Cada vez que um estado é visitado e determinada ação é tomada, o valor é atualizado na posição equivalente ao par estado-ação da matriz. Portanto, conforme estes pares vão sendo visitados, seus valores vão sendo descobertos. Abordagens que representam a função-valor através de tabelas com estados e ações são chamadas tabulares.

Apesar de apresentarem bons resultados para ambientes simples, conforme o número de estados e ações cresce, soluções tabulares apresentam um problema: cada novo par estado-ação precisa ser explorado para ter seu valor descoberto, fazendo com que o treinamento necessário cresça muito rapidamente com o número de estados e ações possíveis. Para casos onde o ambiente possui um número muito grande de estados ou ações, uma solução possível é utilizar aproximação de funções. Métodos de aproximação de fun-

ções funcionam utilizando exemplos de entradas e saídas e produzem uma função que aproxima esse comportamento [SB18, Cap. 9]. Ao utilizar estados ou pares estado ação como entrada e valores como saída, é possível gerar uma aproximação para a função-valor, de forma análoga, mas não idêntica a problemas de regressão em aprendizado supervisionado.

Em soluções com aproximações de funções usualmente se define uma lista de características que descrevem os estados e, para cada uma delas, são definidos pesos e operações realizadas para calcular a saída aproximada. Dessa forma, mesmo que a quantidade de estados cresça, o conjunto de atributos será o mesmo, mudando apenas seus números. Portanto, embora um estado ainda não tenha sido visitado, o algoritmo consegue calcular seu valor com a função-valor aproximada a partir dos pesos calculados com outras experiências exploradas. Em contrapartida, atualizar os pesos da função para melhorar seu comportamento em relação a uma entrada afeta diversos outros estados, sendo necessário encontrar valores que aproximem o melhor possível todos os estados.

Por exemplo, dado um estado caracterizado por N atributos, $x_1(s), x_2(s), ..., x_n(s)$, e uma lista de pesos $w_1, w_2, ..., w_n$, uma forma extremamente simples de aproximar a função-valor seria através de uma função linear, formalizada na equação 2.4.

$$\hat{v}(s, w) = \sum_{i=1}^{N} w_i x_i(s). \tag{2.4}$$

De modo geral, diversos métodos de aproximação de funções podem ser utilizados em aprendizado por reforço. Para isto, são necessárias apenas duas condições: o aprendizado possa ser feito conforme o agente interage com o ambiente e a função possa mudar ao longo do tempo. Esta última condição permite o aprendizado com algoritmos que utilizam predição para determinar o valor futuro dos estados, visto que estes variam conforme o agente é treinado.

2.5 Redes Neurais Artificiais

Redes Neurais Artificiais são um método comumente utilizado para aproximação de funções não-lineares. Elas são compostas por camadas, sendo cada camada formada por uma ou mais unidades chamadas neurônios artificiais. Um neurônio é um modelo que possui elementos como uma lista de entradas, pesos, bias, função de ativação, uma saída e realiza operações com suas entradas para computar o valor da saída.

Um exemplo clássico de neurônio é um Perceptron. Um perceptron com uma lista de entradas x, pesos w e um bias b tem sua saída calculada pela Equação 2.5 [Nie15].

$$y = \begin{cases} 0 & \text{se } w \cdot x + b \leq 0 \\ 1 & \text{se } w \cdot x + b > 0 \end{cases} \tag{2.5}$$

Dada sua arquitetura, ajustar os pesos e bias dos Perceptrons, bem como de neurônios de maneira geral, faz com que sua saída mude. Assim, dentro de certas limitações, um neurônio pode ser treinado para apresentar um comportamento de diversas funções simplesmente modificando os pesos de suas entradas. Na Figura 2.2 é apresentado um neurônio artificial com três entradas.

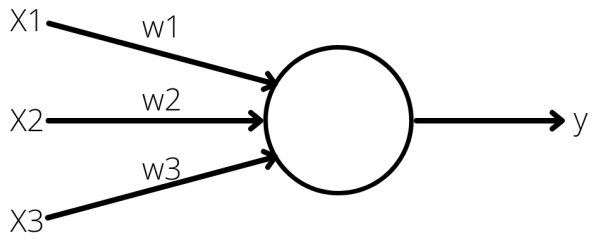


Figura 2.2 – Neurônio Artificial

Diferentes categorias de neurônios possuem diferentes funções de ativação, portanto, comportamentos diferentes. Por exemplo, unidades sigmoides utilizam a função sigmoide para ativação. Sua saída é formalizada na Equação 2.6.

$$y = \frac{1}{1 - exp(-w \cdot x - b)} \tag{2.6}$$

Um tipo de unidade amplamente utilizada, principalmente em camadas ocultas, são as unidades lineares retificadas, do inglês *rectified linear units* (ReLU). Estas unidades utilizam a função de ativação $g(z) = max\{0, z\}$. Uma de suas principais vantagens é que são semelhantes a unidades lineares e fáceis de otimizar. Como a função retorna zero para a metade negativa de seu domínio e apresenta comportamento linear para a outra metade, sua derivada permanece grande quando a unidade está ativa. Já sua derivada de segunda ordem é zero em boa parte do domínio e 1 sempre que a unidade está ativa [GBC16, Cap. 6]. Portanto, métodos de otimização que utilizam derivadas, como o gradiente descendente estocástico, conseguem otimizá-las de forma mais fácil.

Além de diferentes funções de ativação, neurônios podem ser agrupados em camadas e a rede pode ser composta por sequências de diversas camadas, aumentando a complexidade e variedade das funções que a rede pode aproximar. A primeira camada de

uma rede é denominada camada de entrada e a última, camada de saída. Camadas entre a entrada e saída são denominadas camadas ocultas. Cada camada pode possuir múltiplas unidades. Redes em que as saídas de uma camada são utilizadas como entrada apenas das próximas camadas são chamadas *feedfoward* e são a categoria de rede mais comum. Já nos casos onde existem laços de realimentação, são denominadas recorrentes.

Quanto a linearidade, redes neurais *feedfoward* compostas apenas pelas camadas de entrada e de saída com funções de ativação lineares conseguem aproximar apenas funções lineares. Ao adicionar camadas ocultas, porém, a rede torna-se capaz de aproximar funções não lineares [SB18, Cap. 9.7]. Pelo teorema da aproximação universal, uma rede com apenas uma camada oculta com um número grande o bastante de neurônios sigmoides, por exemplo, consegue aproximar qualquer função contínua com qualquer grau de precisão [Cyb89]. Na Figura 2.3 pode-se observar uma rede neural exemplo com 3 entradas, 2 saídas e uma camada oculta composta por 5 unidades.

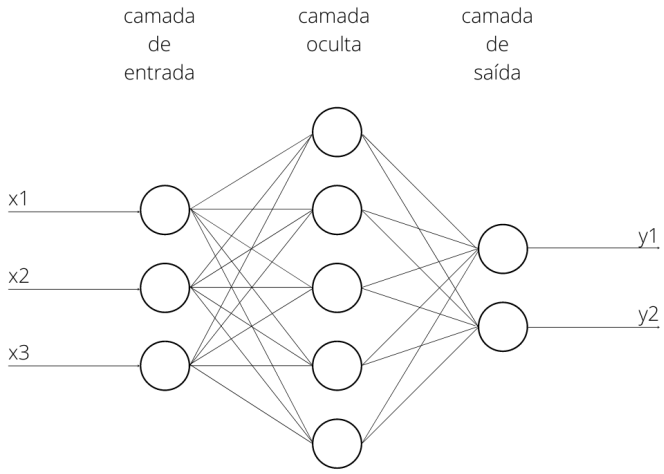


Figura 2.3 – Rede Neural Artificial

O processo de treinamento da rede neural pode ser tratado como um problema de otimização: escolhe-se uma função de perda, também chamada função de custo, e um algoritmo de otimização, que atualiza os pesos da rede para minimizar a perda. Para o treinamento, são necessários dados de entrada e sua saída esperada. Lotes de dados de entrada são utilizados na rede e os resultados são comparados com as saídas esperadas pela função de perda. A função de otimização, então, atualiza os valores dos pesos buscando minimizar o valor da perda. Dentre os algoritmos de otimização, os iterativos baseados em gradiente são bem comuns, principalmente em redes neurais profundas, que serão abordadas na Seção 2.5.1. Estes algoritmos conduzem as funções de custo a valores muito baixos e, diferente dos métodos de tradicionais de resolução de equações lineares, funcionam quando a função de perda não é convexa, como em redes não lineares [GBC16, Cap. 6].

2.5.1 Redes Neurais Profundas

Apesar da sua capacidade de aproximação de funções, redes neurais com apenas uma camada oculta possuem alguns problemas. Primeiro, o teorema da aproximação universal garante a capacidade da rede de aproximação de funções, porém não existe garantia alguma que o algoritmo de treinamento consiga aprendê-la. O segundo ponto a ser considerado é que o teorema menciona que a rede precisa ser grande o bastante para obter qualquer grau de precisão e, com isso, nos piores casos, o número de unidades exponencial pode ser necessário [GBC16, Cap. 6].

Em diversos casos utilizar um número de maior de camadas ocultas pode diminuir o número de unidades necessárias bem como o erro de generalização. De fato, teoria e prática mostram que a utilização de múltiplas camadas ocultas facilita a aproximação de funções complexas, em alguns casos sendo até necessária [SB18, Cap. 9.7].

2.5.2 Redes Neurais Convolucionais

Redes neurais convolucionais são um tipo específico de rede neural focados em dados distribuídos espacialmente em uma ou mais dimensões [GBC16, Cap. 9]. Estas redes são utilizadas, por exemplo, em classificação de imagens, já que estas podem ser representadas como uma matriz de pixels.

Para ser considerada convolucional, uma rede neural deve possuir pelo menos uma camada que efetue operações de convolução. A operação de convolução geralmente é uma operação realizada entre duas funções. No contexto de redes neurais, a convolução ocorre entre a entrada e um núcleo, ou *kernel*. O núcleo é uma matriz com o mesmo número de dimensões que os dados de entrada e um tamanho definido na criação da rede, sendo composto por parâmetros que são adaptados pelo algoritmo de aprendizado. Por fim, o movimento realizado entre o núcleo e a entrada em cada dimensão entre cada operação de convolução é definido por um parâmetro chamado *stride*. Por exemplo, ao realizar convolução em uma imagem, que contém duas dimensões, usando *strides* 1×1, a cada operação de convolução será feito o deslocamento de apenas um píxel.

Os dados utilizados em redes neurais convolucionais possuem, além de dimensões, um ou mais canais. Cada canal representa a observação de uma quantidade diferente em determinado ponto no tempo ou espaço [GBC16, Cap. 9]. Em imagens, por exemplo, os dados podem ser tratados como duas dimensões, altura e largura, e três canais, cada um indicando a quantidade de vermelho, azul e verde de determinado píxel.

Algumas características importantes de camadas convolucionais são conexões esparsas, compartilhamento de parâmetros e representações equivariantes. Em redes neu-

rais que utilizam multiplicação de matrizes, todas as unidades de saída interagem com todas as unidades de entrada. Em contrapartida, redes neurais convolucionais onde o núcleo é menor que a entrada, são conectadas à unidade da saída apenas as unidades afetadas pela convolução com o *kernel*. Quanto ao compartilhamento de parâmetros, como em redes convolucionais cada parte do núcleo é utilizada em operações com praticamente todas as posições da entrada, muitos parâmetros são compartilhados. Por fim, a propriedade da equivariancia diz que mudanças na entrada geram mudanças do mesmo tipo na saída [GBC16, Cap. 9].

Outro ponto diferencial destas redes é sua capacidade de tratar entradas de diferentes tamanhos: ao tratar duas imagens de largura e/ou altura diferente, seria extremamente difícil modelar uma rede neural tradicional. Já no caso de redes convolucionais, basicamente a diferença é que a convolução com o núcleo será aplicado mais ou menos vezes, dependendo do tamanho da imagem, e ajustará a saída de acordo. Caso a saída da rede tenha um tamanho fixo, é necessário realizar algumas mudanças extras na arquitetura para ajustar o tamanho da saída para o correto.

2.6 Q-learning e Deep Q-learning

Q-learning é um algoritmo *off-policy* de aprendizado, como mencionado na Seção 2.2.3, que utiliza diferença temporal. Algoritmos de diferença temporal utilizam a observação da próxima etapa para fazer a estimativa de valores futuros, utilizando a função-valor atual, para a atualização do valor do estado. *Q-learning* aproxima a função-valor aprendida, Q, da função-valor ótima, q_*, independente da política seguida [SB18, Cap. 6.5]. Contudo, para que a convergência de Q para q_* ocorra, é necessário garantir que todos os pares estado-ação sejam visitados.

O algoritmo de *Q-learning* é formalizado pela Equação 2.7, sendo $Q(S_t, A_t)$ é o valor da ação A_t em um estado S_t, R_{t+1} a recompensa do estado destino e $\max_a Q(S_{t+1}, a)$ o valor da melhor ação possível do novo estado, segundo a função Q. O fator $\alpha \in (0, 1]$ define o quanto a experiência atual afeta o valor antigo da função para o par $\langle S_t, A_t \rangle$. Já o coeficiente $\gamma \in [0, 1]$ é o fator de desconto de recompensa futuras. Os valores da função Q para cada par estado-ação são denominados *Q-values*.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)], \tag{2.7}$$

Deep Q-learning ou *Deep Q-Networks* (DQN) é uma versão do algoritmo de *Q-learning* adaptada para utilização de redes neurais artificiais profundas para aproximar a função-valor. Este algoritmo aproxima a função Q através de uma rede neural, chamada *Q-network*. São duas as principais diferenças entre o algoritmo clássico e o *Deep Q-learning*:

como são selecionados os dados para o treinamento e a aproximação dos valores utilizados na atualização da rede.

Enquanto a versão tabular atualiza os estados conforme vão sendo explorados, redes neurais fazem atualizações utilizando lotes de dados. Utilizar um lote de dados composto por uma sequência de experiências, porém, pode resultar em dados com altos níveis de correlação, uma vez que que o estado atual impacta diretamente nos estados futuros possíveis. Para evitar isso, o algoritmo de *Deep Q-learning* utiliza experiências passadas selecionadas aleatoriamente: as tuplas $\langle s, a, r, s' \rangle$, sendo s o estado no momento da tomada de decisão, a a ação executada, s' o estado resultante e r sua recompensa, são armazenadas em uma memória com determinado tamanho máximo e, no momento da atualização da rede neural, um lote de experiências aleatórias é selecionado desta memória. Conforme a memória atinge seu tamanho máximo, experiências mais antigas são descartadas para darem lugar às novas [MKS+15]. Portanto, conforme o número de dados nessa memória aumenta, menor a probabilidade de duas experiências aleatórias de uma mesma sequência serem selecionadas, assim diminuindo a correlação dos dados de treinamento.

Quanto à aproximação dos valores utilizados para atualização da função, na versão tabular a mesma função sendo atualizada é utilizada nas predições dos *Q-Values*. Na versão com redes neurais profundas, por se tratar de aproximação de funções não lineares, o processo de treinamento pode implicar em instabilidade na rede durante o processo. Portanto, para contornar este problema, utiliza-se uma segunda rede neural para realizar as predições dos valores utilizados na atualização, chamada rede alvo. Esta rede é basicamente uma cópia da *Q-network* e tem seus pesos atualizados periodicamente com os pesos da rede treinada [MKS+15].

2.7 OpenAi Gym

OpenAI Gym é um conjunto de ferramentas para desenvolvimento e comparação de algoritmos de aprendizado por reforço. A plataforma, desenvolvida na linguagem *Python*, apresenta uma série de ambientes com uma interface padronizada, facilitando o processo de desenvolvimento e validação de agentes de aprendizado por reforço mais genéricos. Através dela é possível, por exemplo, executar o mesmo agente em múltiplos ambientes com poucas adaptações do ponto de vista do ambiente, visto que a interface de comunicação com eles será a mesma.

A plataforma é focada em aprendizado por reforço baseado em episódios, sendo um episódio um ciclo onde o ambiente parte de um estado inicial e uma sequência de observações e ações são tomadas pelo agente até que o ele atinja estado terminal [BCP+16].

A interface com o ambiente provida pelo *Gym* conta com funções para reiniciar o ambiente, que retorna uma observação inicial, listar ações disponíveis e executar uma

ação, que retorna uma observação do resultado, sua recompensa e um valor indicando se o ambiente atingiu o estado terminal. Portanto, a ferramenta funciona como uma interface padrão de sensores e atuadores do ambiente, como abordado na Seção 2.2.1, permitindo que o agente receba observações e execute ações.

Gym conta com diversos ambientes já implementados, sendo sobre controle clássico, jogos do console *Atari*, simuladores de robôs 2D e 3D, algoritmos de problemas matemáticos, jogos de tabuleiro, dentre outros. Além destes, é possível desenvolver ambientes customizados para a ferramenta. Para isto, é necessário realizar algumas configurações e implementar métodos para controlar o ambiente, mapear observações e ações possíveis e agir sobre ele. Neste trabalho foi utilizada esta opção de criação de ambientes customizados e implementou-se um ambiente do *Gym* para o *Battle for Wesnoth*.

3. DESENVOLVIMENTO

O desenvolvimento deste trabalho se deu em duas principais etapas: a implementação de um ambiente do *OpenAI Gym* para o jogo *Battle for Wesnoth* e a implementação de um agente de aprendizado por reforço para o jogo utilizando este ambiente. O ambiente foi desenvolvido para executar o jogo e comunicar-se com ele, fornecendo observações sobre a partida e permitindo a execução de ações. O agente utiliza essas observações para escolher a ação das unidades em cada momento e aprender baseado no resultado de suas escolhas. O código das implementações realizadas, tanto para o ambiente do *OpenAI Gym* quanto para o agente de aprendizado por reforço, pode ser encontrado em: https://github.com/tgambim/wesnoth-rl-agent.

3.1 Ambiente OpenAI Gym

O ambiente do *OpenAI Gym* foi desenvolvido de modo a oferecer uma interface padronizada para comunicação com o jogo. Para isso foi necessário o desenvolvimento de um *add-on* para o jogo, que permite comunicação com *scripts* externos, e a implementação de um programa em *Python* para executar o jogo e manipulá-lo. O fluxo implementado pode ser observado na Figura 3.1

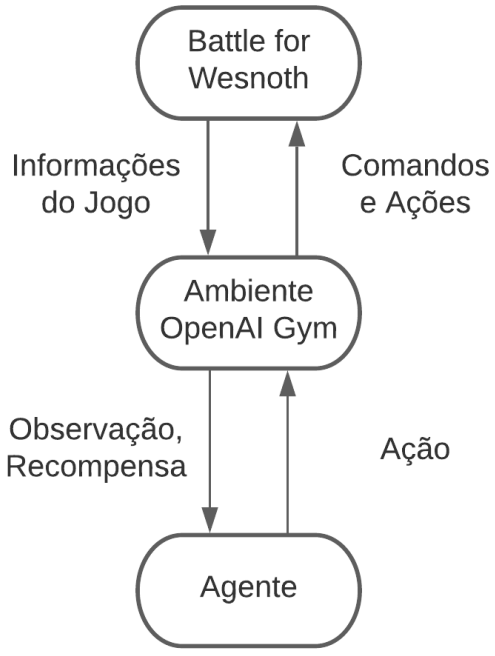


Figura 3.1 – Diagrama de funcionamento da integração com o *Battle for Wesnoth*.

3.1.1 Ambiente em Python

O ambiente em *Python* do *OpenAI Gym* foi implementado conforme as especificações de ambientes customizados disponibilizadas pela *OpenAI* [Ope21], contando com as funções: executar ação, tendo como resposta uma nova observação, recompensa e indicação de finalização do jogo; reiniciar o jogo, tendo como retorno uma observação inicial da partida e, por fim, finalizar o jogo. Na Figura 3.2 é apresentado o diagrama de UML da implementação realizada para o ambiente *Gym*.

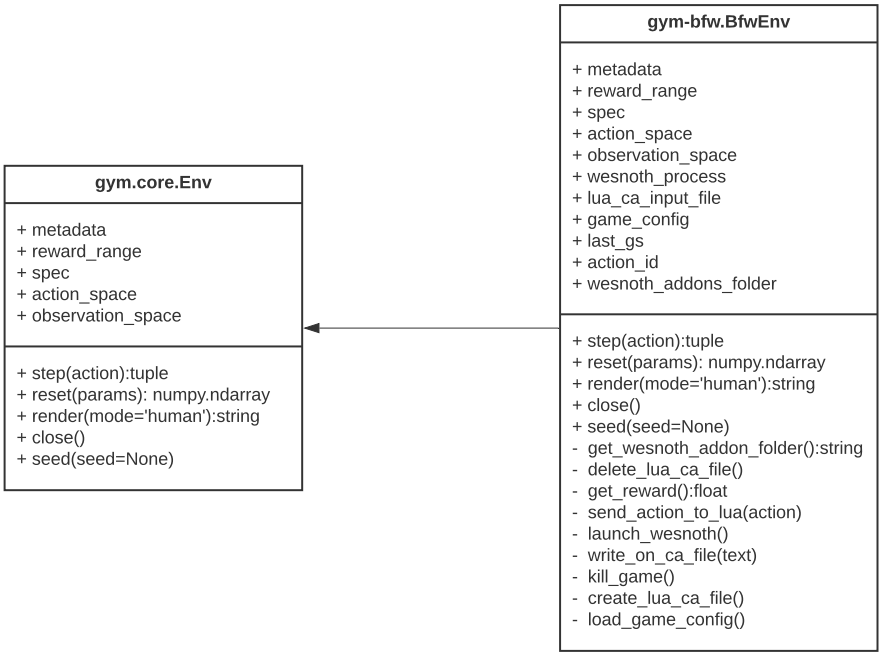


Figura 3.2 – Diagrama de UML da implementação do ambiente em *Python*.

O ambiente jogo foi implementado para funcionar conforme as seguintes regras:

1. Antes da primeira utilização e após cada partida deve ser executada a função *reset*.
2. Para execução das ações deve ser utilizada a função *step*, passando como parâmetro a tupla da ação escolhida.
3. Para finalizar um jogo em andamento deve ser utilizada a função *close*.

Inicialmente, ao ser instanciado, o ambiente do *Gym* executa um processo do jogo passando como parâmetro uma opção para ele apenas retornar o diretório onde ficam armazenados os *add-ons* e salva o retorno. Essa informação é necessária, pois nesse diretório são criados os arquivos utilizados para o envio de informações do *script Python* para a

IA customizada do jogo. Pela saída do processo são enviadas as informações do jogo para o ambiente.

A função que reinicia o ambiente foi desenvolvida para iniciar e sincronizar um processo do jogo. Inicialmente, a função abre um novo processo passando parâmetros para remover a interface, o som e atrasos do jogo, iniciá-lo em uma partida multijogador, repeti-la por 1.000.000 vezes, e encerrar o processo depois da última repetição. O parâmetro da repetição é importante, pois faz com que a partida seja reiniciada após sua finalização, o que é muito mais rápido que encerrar o processo e abri-lo novamente. Por fim é passado um parâmetro indicando que o cenário do jogo será o cenário de teste criado no *add-on*. Uma vez iniciado o processo, lê-se a saída do jogo até que seja recebido o nome do arquivo que será utilizado para comunicação. Ao recebê-lo, o *script* então cria um arquivo na pasta de *add-ons* com este nome. Por fim, a função espera até receber as configurações do jogo, contendo o tamanho do mapa, o estado inicial do jogo e as ações disponíveis no primeiro turno, e retorna uma observação contendo as informações da partida extraídas do estado do jogo.

O método *step* é responsável por executar uma ação e retornar novas informações sobre o jogo. Ele começa validando a ação escolhida: caso seja inválida ele retorna a mesma observação, uma recompensa de -15 e a sinalização indicando que o jogo ainda não acabou; caso seja válida, o ambiente então escreve as informações da ação no arquivo de comunicação. Após o envio da ação escolhida, o programa espera até receber a próxima observação e as próximas ações disponíveis ou uma mensagem de jogo finalizado. Por fim, a função retorna a nova observação, a recompensa, um indicador de jogo finalizado e, nas informações extras, um indicador de novo turno.

Para o cálculo da recompensa foram testadas duas abordagens. A primeira delas foi a utilização de -1 para derrota, +1 para vitória e 0 para todas as outras situações. A segunda dá as mesmas recompensas, com a diferença que ações resultando em captura de aldeias e morte de unidades inimigas recebem recompensa igual a 0,05, recrutamento de unidades igual a 0,1 para cada nova unidade e -0,02 caso a unidade tenha sido morta na última ação.

Quanto às observações realizadas a partir do estado do jogo, estas foram representadas como matrizes tridimensionais com as duas primeiras dimensões correspondendo ao tamanho do mapa, e a terceira sendo igual à 8. Assim, para cada hexágono do mapa foram extraídos, respectivamente, os atributos: categoria de terreno, da construção, jogador dono da aldeia, jogador dono da unidade, unidade líder, categoria e pontos de vida dela, movimentação disponível. Todas as informações, com exceção do terreno, que existe obrigatoriamente para todas as coordenadas, utilizam o valor zero caso o recurso em questão, unidade ou construção, não exista naquele hexágono. Para representação de jogador dono do recurso foram utilizados os valores -1 e 1 indicando inimigos e aliados, respectivamente. Para os dados nominais foram definidos números para cada opção. Por fim, todos os dados

da matriz foram normalizados. A representação matricial conforme o mapa foi escolhida para possibilitar o uso de redes neurais convolucionais na implementação do agente.

No que se refere à representação das ações, estas são implementadas por uma tupla contendo coordenadas de origem e destino, a categoria de ação (0 para movimento, 1 para ataque ou 2 para recrutamento), bem como as coordenadas e pontos de vida restantes do alvo em caso de ataque. Todas as informações referentes a coordenadas são indexadas a partir de zero.

3.1.2 Add-on

O *add-on* para o jogo consiste em uma IA customizada e um cenário de teste utilizado nas simulações. O cenário implementado possui uma campanha de um mapa com dois jogadores, ambos da mesma facção, sendo um controlado pela IA padrão do jogo e o outro pela IA customizada implementada. Ambos os jogadores começam com uma unidade líder do mesmo tipo e um total de 40 unidades de ouro. Os mapas utilizados foram espelhados de forma que ambos os jogadores tenham as mesmas distâncias entre terrenos e construções a partir da posição inicial de sua unidade líder, além de não possuírem névoa de guerra. O objetivo foi que ambos os jogadores tenham as mesmas condições de jogo. Quanto à IA customizada, esta foi implementada na linguagem Lua conforme especificado na *wiki* do *Battle for Wesnoth* [Wes21]. A IA implementada consiste em apenas uma *Candidate Action* (CA), como abordado na Seção 2.1, responsável pela comunicação com o programa em *Python*. Todas as CAs padrões do jogo foram removidas dessa IA customizada, fazendo com que a única utilizada seja a implementada para a comunicação com o *script* externo. Na Figura 3.3 é apresentado o diagrama UML da *Candidate Action* desenvolvida.

A comunicação entre o ambiente do *Gym* e a CA ocorre com as informações sendo enviadas a partir da CA através da saída padrão do processo do jogo e as respostas enviadas pelo *script* externo através de um arquivo do tipo Lua. O jogo é iniciado diretamente no cenário de testes criado e, assim que o turno do jogador controlado pela IA customizada começa, a CA que se comunica com o *script* externo é chamada. Na primeira execução da função de avaliação, a *Candidate Action* cria uma função para tratar o evento de fim de jogo e um *hash* aleatório para ser utilizado como nome do arquivo de comunicação. A função de fim de jogo é disparada sempre que a partida se encerra, verifica se o jogador controlado pela IA perdeu sua unidade líder para determinar se foi uma vitória ou uma derrota, e envia uma nova observação do jogo seguida pelo resultado para o *script* externo. Já o nome do arquivo de comunicação é enviado para o programa em *Python* assim que é gerado e, após isso, a IA fica aguardando a criação do arquivo com esse nome da pasta de *add-ons*.

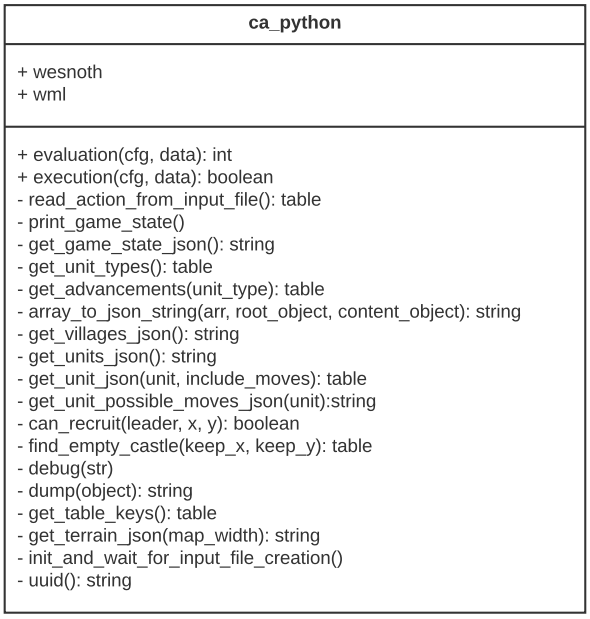


Figura 3.3 – Diagrama de UML da *Candidate Action* implementada.

Este processo de comunicação e sincronização inicial entre o ambiente do *Gym* e o *add-on* foi sugerido inicialmente no fórum do jogo pelo usuário *Mattsc* [Mat21] e implementado por Dominik Stelter [Ste21]. O restante da integração, como mapeamento do estado do jogo, ações disponíveis, formatação da comunicação, execução de ações, dentre outros fatores, são uma implementação própria criada para suprir as necessidades do processo de aprendizado do agente.

Com exceção da primeira execução, que faz essa criação do arquivo antes do fluxo padrão, todas as execuções da CA possuem o seguinte comportamento:

1. Inicia-se a função de avaliação da CA.
2. Executa-se uma checagem de quantas unidades ainda têm ações restantes. Caso não exista nenhuma, a avaliação retorna 0, indicando finalização do turno. Caso contrário, a execução continua.
3. Envia-se para o programa externo o estado do jogo, seguido pela lista de ações disponíveis.
4. Realiza-se a leitura do arquivo de comunicação até que uma nova ação seja detectada nele.
5. Armazena-se a ação escolhida na variável de dados da CA para utilização posterior na função de execução. A função de avaliação retorna 999.990. Como a CA imple-

mentada é a única da IA customizada, 999.990 será a melhor avaliação e, então, a função de execução da CA será chamada.

6. A função execução carrega a ação selecionada e executa-a.

Esta sequência é executada repetidamente durante todos os turnos até o final da partida.

Cada mensagem enviada do *add-on* para o ambiente *Python* inicia com uma marcação indicando o tipo de mensagem, seguida por seu conteúdo. Com exceção da mensagem inicial com o nome do arquivo de comunicação, cujo conteúdo é apenas um texto, todas as outras mensagens são codificadas no formato JSON, facilitando assim seu processamento em programas desenvolvidos em outras linguagens de programação, tendo em vista que boa parte das linguagens contam com funções nativas ou alguma biblioteca para codificação e decodificação de JSON.

Os dados enviados sobre o estado do jogo são divididos em quatro principais categorias: mapa, turno, jogo e jogador. A primeira contém informações sobre o mapa, como dimensões, horário, aldeias, terrenos e unidades. A segunda lista apenas o turno atual e o turno limite. Jogo conta com dados sobre todas as categorias de unidades que podem aparecer na partida e uma indicação de final de jogo. As informações do jogador envolvem o identificador do lado do jogador, ouro, número de aldeias e número de unidades. Cada unidade listada no mapa possui informações como localização, tipo, vida, movimentos e ataques totais e restantes, efeitos positivos e negativos, nível e uma lista de ações disponíveis.

Quanto à lista de ações de cada unidade, esta contém três principais informações: as duas coordenadas de destino e o tipo da ação, que pode ser ataque, recrutamento ou movimento. Em caso de recrutamento, é enviado ainda a categoria de unidade a ser recrutada e seu custo. Em caso de ataques, são adicionadas informações sobre as coordenadas do alvo, sua vida máxima e atual, a arma utilizada e a vida média, tanto do alvo quanto da unidade atacante, baseada em uma simulação de combate. Para simplificar o problema a ser resolvido pelo agente, adicionou-se também uma ação de ataque que seleciona a melhor arma automaticamente e uma ação de recrutamento que recruta a melhor unidade para cada situação. A seleção da melhor arma acontece automaticamente quando nenhuma arma é passada para a função de ataque. Já o recrutamento automático foi feito utilizando a CA genérica de recrutamento disponibilizada com as *Candidate Actions* padrão do jogo e considera o estado atual do jogo e variáveis como vida máxima, custo, tipos das unidades inimigas, movimento, dentre outros fatores.

A execução de ações é efetuada de maneira bem simples. Primeiramente é executado a parte de movimento da ação, caso as coordenadas de origem diferirem do destino. A segunda etapa é sinalizar para o jogo que o movimento daquela unidade acabou. Por fim, se for uma ação de recrutamento ou ataque, esta parte é executada.

3.2 Agente de Aprendizado por Reforço

O agente de aprendizado por reforço foi inicialmente implementado usando um método tabular e, posteriormente, foram implementadas versões utilizando aproximação de funções com funções lineares e não lineares.

Em todas as implementações, o treinamento começa com um fator de exploração ϵ igual à 100% que vai decrescendo até um valor mínimo de 1%. Assim, no início do treinamento o agente executa praticamente todas as ações de forma aleatória para descobrir o maior número de recompensas. Conforme o treinamento avança, o fator de exploração é diminuído de forma à descobrir mudanças que possam melhorar o retorno da política. Foram testadas algumas variações de funções para gerar os valores de ϵ ao longo do treinamento, que podem ser encontradas no Capítulo 4.

Devido à possibilidade de movimento para qualquer hexágono desocupado no alcance, unidades têm uma grande lista de possibilidades de ações de movimento. Em contrapartida, ações de ataque precisam de um alvo adjacente e as de recrutamento devem ser realizadas pela unidade líder em uma torre de menagem, o que impacta em um número reduzido de ações destes tipos. Portanto, escolher uma ação aleatória do espaço de ações acaba tendo uma probabilidade maior de resultar em uma ação de movimento em relação às outras duas. Devido à importância de ações de ataque e recrutamento, implementou-se um balanceamento nessa escolha: as ações foram divididas em três grupos, um para cada categoria de ação, e cada grupo recebeu uma probabilidade igual de escolha. Quando um grupo é escolhido, uma ação aleatória dele é utilizada.

3.2.1 Implementação Tabular

A implementação do agente de aprendizado por reforço utilizando um método tabular foi realizada com o algoritmo *Q-learning*. Este algoritmo foi escolhido, porque utiliza diferença temporal e possui uma implementação simples.

A grande dificuldade em criar um agente tabular para o jogo está em escolher quais parâmetros definem os estados e ações: caso muitos fatores sejam considerados, o espaço de estados e o número de ações cresce muito rapidamente. O problema disto é que algoritmos tabulares precisam explorar todos seus estados e ações para convergirem para a política ótima. Portanto, quanto maior o espaço de estados e as possíveis ações, será necessário maior treinamento para que todos os pares estado-ação sejam visitados, inviabilizando a utilização do agente.

Para mitigar essa situação, foram definidos alguns atributos para determinar estados e ações. Para estados, os atributos escolhidos foram:

1. vida da unidade: abaixo de 50% ou acima de 50%.
2. líder: indica se a unidade é uma unidade líder.
3. tipo de construção: indica se a construção onde a unidade está localizada é uma aldeia, torre de menagem ou nenhuma destas.
4. ouro: quantidade de ouro do jogador, podendo ser maior que 20, entre 20 e 0 e menor que 0.
5. renda: renda em ouro por turno do jogador. Os intervalos considerados foram: menor ou igual à 0, 1 até 2, 3 até 4 e maior que 4.
6. unidades: diferença de unidades entre os jogadores. Pode assumir os intervalos: menor que -4, -4 à -3, -2 à -1 , 0 , 1 à 2, 3 à 4 e maior que 4.

Do mesmo modo, para as ações foi necessário a seleção de alguns atributos. Foram eles:

1. tipo da ação: ataque, recrutamento ou movimento
2. tipo do construção: tipo de construção no destino do movimento. As opções consideradas são torre de menagem, aldeia aliada, aldeia inimiga ou nenhuma.
3. vida do alvo: em caso de ataque, indica se a vida do inimigo está abaixo de 50% ou acima de 50%.
4. alvo líder: em caso de ataque, indica se o inimigo é um líder.
5. torre de menagem: distância da torre de menagem mais próxima.
6. inimigo: distância até o inimigo mais próximo.
7. aldeia inimiga: distância até a aldeia inimiga mais próxima.
8. líder inimigo: distância até o líder inimigo.

Para as medidas de distância, é calculada a distância euclidiana entre as coordenadas de origem e destino, e realizada uma divisão pelo número de ações de movimento da unidade, arredondando o resultado para cima. Com isso, tem-se uma aproximação do número de turnos necessários para a unidade chegar até o alvo. Como valores possíveis, assumiu-se 0, 1 e maior que 1. Como os mapas testados foram pequenos, a maioria das unidades consegue atravessar o mapa em apenas dois turnos, então estes intervalos de distâncias são o suficiente.

Com os atributos de estados e ações definidos, criou-se um identificador para cada possibilidade de estados, ações e a tabela de *Q-Values*. O agente, para escolher sua ação,

determina seu estado extraindo os atributos selecionados a partir da observação do ambiente e seleciona a ação possível com maior *Q-Value* a partir do estado atual. A atualização do *Q-Value* na tabela após uma experiência é efetuada conforme a Equação 2.7.

3.2.2 Implementação com Aproximação de Função

No desenvolvimento do agente com aproximação de funções, tanto lineares quanto não lineares, utilizou-se uma abordagem com redes neurais. Apesar da possibilidade de utilizar outros métodos, principalmente na versão linear, optou-se por essa alternativa pela facilidade de transição entre a versão linear para a não-linear, bastando modificar a arquitetura da rede. Como algoritmo, optou-se por *Deep Q-learning* por sua simplicidade e semelhanças com a versão tabular.

Devido à utilização de aproximação de funções com redes neurais, realizar treinamentos com experiências individuais não é uma opção viável. Passou-se a utilizar uma memória para armazenar as experiências referentes a cada etapa da partida, sendo uma etapa a observação e execução de ação de uma unidade. No processo de treinamento, então, lotes aleatórios são selecionados dessa memória e utilizados para atualizar a rede neural. Nos testes efetuados, utilizou-se uma memória armazenando até 50.000 etapas, e após isso as entradas mais antigas são descartadas ao inserir as novas. Cada lote de dados utilizado no treinamento possui 32 amostras aleatórias de experiências retiradas dessa memória. Por fim, a quantidade de jogos com derrota é muito maior que a de vitórias na memória, principalmente em estágios iniciais do treinamento em que o fator de exploração é muito alto. Portanto, forçou-se a seleção de pelo menos 1 amostra, dentre essas 32, com recompensa igual à 1, indicando vitória, em cada lote de treinamento. Após alguns treinamentos de teste essa abordagem mostrou melhores resultados que a versão com 32 amostras totalmente aleatórias.

O processo de treinamento do agente consistiu na execução de 20.000 partidas, e após cada uma delas foi efetuado treinamento da rede neural. Como o número de turnos e, como consequência, ações executadas em cada partida varia muito conforme o valor de ϵ, foi definido um número de treinamentos por partida baseado na quantidade de experiências obtidas no jogo: para cada etapa foi efetuado um treinamento com um lote de dados. Por exemplo, após uma partida com 10 ações, são executadas 10 seções de treinamento. Para manter a estabilidade da rede neural utilizou-se uma rede secundária no treinamento, a rede alvo, como especificado na Seção 2.6, e seus pesos foram atualizados com os pesos da *Q-Network* a cada 500 partidas.

Para criação e treinamento da rede neural utilizou-se a biblioteca *TensorFlow* e a API *Keras*. *TensorFlow* é uma biblioteca de aprendizado de máquina com uma ampla gama de aplicações, incluindo redes neurais. *Keras* é uma API de aprendizado profundo que,

por seu alto nível, permite a criação de redes neurais de maneira intuitiva. Dentre outras bibliotecas, *Keras* pode ser utilizado com *TensorFlow* na criação de redes neurais.

O aprendizado da rede foi configurado utilizando um fator α igual a 1%. Como as recompensas mais importantes, que representam vitória (1) e derrota (-1), são dadas somente no final da partida, definiu-se o fator γ igual a 99%, de modo que as recompensas futuras tenham um grande impacto nas ações. Como função de perda foi utilizada a função perda de Huber. Esta função possui um comportamento combinado das funções de erro quadrático médio e erro absoluto médio, utilizando a primeira para valores de perda pequenos e a segunda para valores de perda muito grandes. Por fim, para atualização dos pesos foi utilizado o algoritmo de otimização Adam, um algoritmo substituto ao gradiente descendente estocástico que apresenta bons resultados com redes neurais profundas.

A entrada da rede foi uma matriz contendo informações sobre o estado do jogo e uma ação possível a ser executada. Esta matriz é obtida concatenando a matriz de observação retornada pelo ambiente com uma indicando a ação a ser executada. A matriz de ação possui as duas primeiras dimensões iguais ao mapa do jogo, assim como a de observação, e em sua terceira dimensão, de tamanho 3, possui, respectivamente, a marcação da coordenada de origem da unidade utilizada na ação, uma indicação da categoria de ação na coordenada de destino e, por fim, o novo valor de vida do alvo, de acordo como uma simulação, em caso de ataque. Assim, a matriz de estado-ação utilizada como entrada da rede neural possui as duas primeiras dimensões iguais à largura e altura do mapa e uma terceira dimensão de tamanho 11. A saída da rede, tanto no caso linear quando não linear, possui tamanho igual a um e indica o *Q-Value* do par estado-ação utilizado como entrada.

A versão linear foi desenvolvida utilizando uma rede de apenas duas camadas, entrada e saída, totalmente conectadas. A entrada inicialmente é re-arranjada em apenas uma dimensão na camada de entrada. Sua saída é uma camada ativada por uma função linear e possui um neurônio. Utilizando um mapa de entrada com tamanho 10×10, a arquitetura da rede pode ser observada na Figura 3.4.

A versão não-linear foi implementada utilizando uma rede neural de cinco camadas. A entrada é inserida em uma camada convolucional de duas dimensões, com 32 canais, tamanho do núcleo igual a 4×4, *strides* 1×1 e utiliza a função ReLU como ativação. As duas camadas seguintes são também camadas convolucionais de duas dimensões, ambas com 64 canais, *strides* 1×1, função de ativação ReLU e com tamanho do núcleo igual à 3×3 e 2×2, respectivamente. A saída da última camada convolucional é então re-arranjada em uma dimensão e a quarta camada é uma totalmente conectada de 512 neurônios com função de ativação ReLU. Por fim, a quinta camada é uma camada totalmente conectada com apenas um neurônio e função de ativação linear. Na Figura 3.5 é apresentado um diagrama da arquitetura implementada, utilizando como exemplo um mapa de tamanho 10×10.

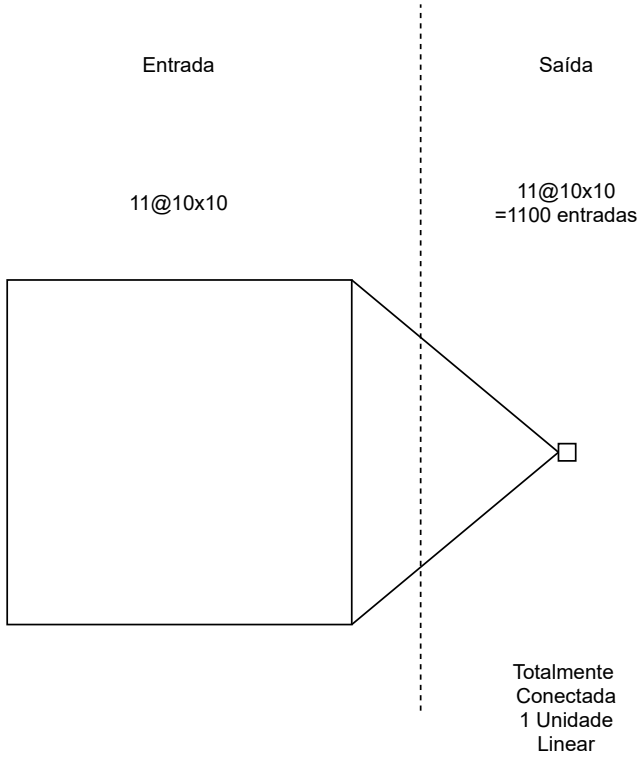


Figura 3.4 – Diagrama de rede neural na versão linear.

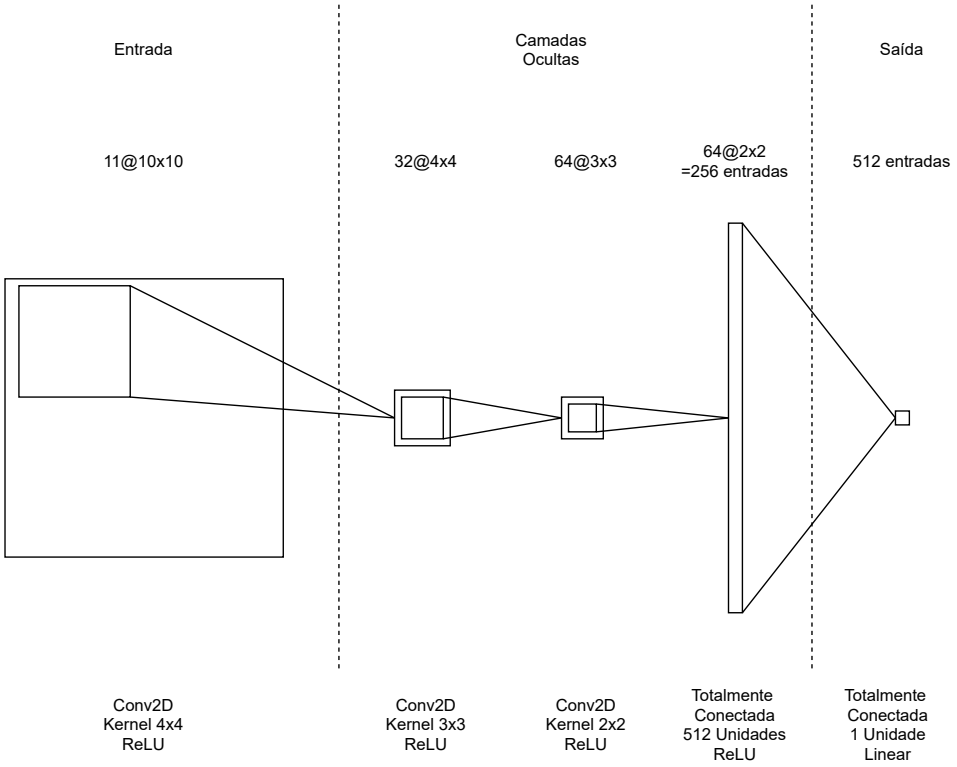


Figura 3.5 – Diagrama de rede neural na versão não-linear.

4. RESULTADOS

Para todas as versões implementadas, tanto o processo de treinamento quanto os testes foram repetidos cinco vezes, de modo a obter um comportamento médio do agente desenvolvido. Como métrica de análise dos resultados, utilizou-se a recompensa média das partidas e o número de vitórias. Para uma melhor visualização do crescimento da taxa de vitórias, os resultados foram agrupados em sequências de 100 partidas e calculadas a recompensa média e a taxa de vitórias de cada conjunto de partidas.

No processo de treinamento foi utilizado um fator de exploração ϵ decrescente no decorrer dos episódios. Iniciou-se com uma taxa constante para decrementar *epsilon*, fazendo-o variar de 1 no episódio inicial à 0,01 no episódio final. Porém, partidas com o fator de exploração muito alto normalmente acabam de maneira rápida, poucas ações são exploradas e raramente resultam em vitórias. Dessa forma, com ϵ muito alto a memória de repetição contém majoritariamente sequências de ações semelhantes e que resultam em derrota. Já com o fator de exploração menor, além de os jogos durarem mais e novos comportamentos serem descobertos, como a conquista de aldeias que gera ouro e possibilita recrutamento, a taxa de vitórias aumenta, fazendo com que uma maior variedade de sequencias de ações que resultam em vitória sejam inseridas na memória de treinamento. Foram realizados alguns testes com diferentes funções para gerar valores de ϵ. Dentre os testes realizados, a função que apresentou melhores taxas de vitórias e recompensa média em um menor número de episódios é formalizada na Equação 4.1, sendo n o número de episódios totais e x o episódio atual. Com essa equação, o valor de ϵ inicia em 1 decai rapidamente nos primeiros 40% dos episódios totais, atingindo um fator de exploração igual à 0,2. Nos 60% restante dos episódios, ϵ decai de forma mais lenta, chegando em 0 no último episódio do treinamento.

$$\epsilon = \begin{cases} 1 - x * \frac{2}{n} & \text{se } x \leq 0,4 * n \\ 0.20 - (x - 0,4 * n) * \frac{0.2}{0,6 * n} & \text{se } x > 0,4 * n \end{cases} \tag{4.1}$$

Algumas variações foram testadas tanto na implementação do agente quanto do ambiente. No ambiente as variações se restringiram às recompensas. A primeira abordagem considerou apenas recompensas para o resultado da partida, sendo elas -1 para derrotas e 1 para vitórias. A segunda implementação utilizou, além das recompensas de resultado da partida, recompensas para:

1. Recrutamento: 0,1 por unidade recrutada.
2. Conquista de aldeia: 0,05 por aldeia conquistada.
3. Matar unidade inimiga: 0,05 por unidade.

4. Morte da unidade aliada: -0,02.

As variações no agente foram efetuadas na forma como as unidades são tratadas. Duas abordagens foram analisadas: ações futuras considerando todas as unidade e considerando apenas a unidade em questão. A ideia da primeira implementação usou as ações das próximas unidades para contabilizar a recompensa futura de uma ação, de modo que o agente aprendesse como uma unidade impacta nas ações das outras em um turno. A segunda abordagem considera apenas as ações que a própria unidade poderá realizar no turno seguinte na hora de aproximar sua recompensa futura. Das abordagens testadas, a implementação que obteve maior sucesso após o treinamento, apresentando uma maior taxa de vitórias e cujo resultado será discutido neste trabalho, foi a implementação utilizando recompensas para pequenos objetivos e considerando as ações de cada unidade individualmente.

4.1 Versão Tabular

Para a versão tabular, foram executados treinamentos de 40.000 partidas. Devido à necessidade de visitar todos os estados, foi necessário um número de partidas maior do que as versões com aproximação de funções. Apesar do número maior de partidas utilizadas no treinamento, devido às simplificações feitas na escolha dos atributos para definição dos estados, o agente não obteve bons resultados, chegando a uma taxa de vitórias pouco melhor que a versão aleatória. Uma possível solução seria adicionar mais atributos para descrever o estado, porém, como o ambiente possui uma grande complexidade, adicionar todos os atributos necessários para melhorar os resultados implicaria em um número muito grande de pares estado-ação, tornando necessário a execução de muitas partidas para o treinamento. Portanto, optou-se por implementar uma nova versão com aproximação de funções para contornar esse problema.

4.2 Versão Linear

Com a rede neural linear, foram executados cinco treinamentos com 20.000 partidas cada. Na Figura 4.1 são apresentados os resultados médios dos cinco treinamentos. Já nas Figuras 4.2 e 4.3 são apresentados o melhor e pior resultado, respectivamente, considerando os dados dos últimos 500 episódios de treinamento.

Para verificar os resultados das redes treinadas de maneira mais precisa, foram executados 5 testes, cada um composto por 500 partidas, em ambas as redes com melhor e pior resultado ao final do treinamento. Neles utilizou-se um fator de exploração zero e

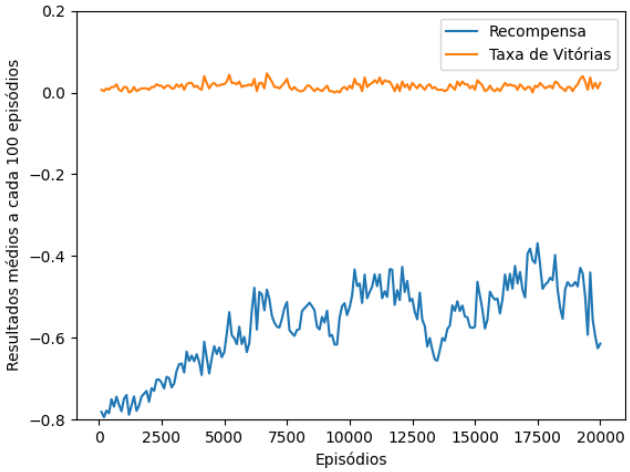


Figura 4.1 – Média dos resultados dos cinco treinamentos da versão linear.

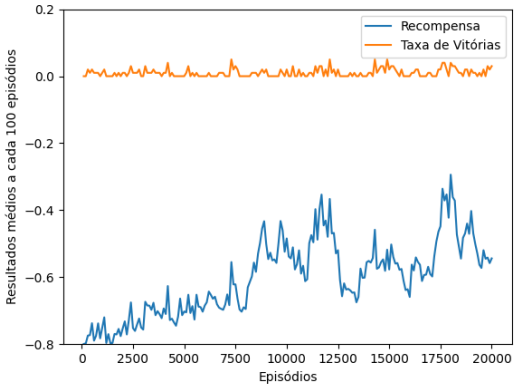


Figura 4.2 – Melhor resultado dos cinco treinamentos da versão linear.

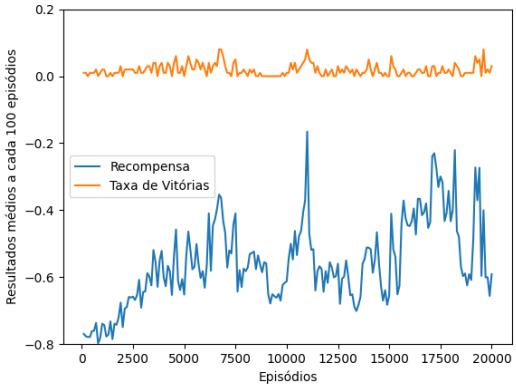


Figura 4.3 – Pior resultado dos cinco treinamentos da versão linear.

não foi efetuado nenhum treinamento durante a execução. Os resultados obtidos podem ser verificados nas Figuras 4.4 e 4.5.

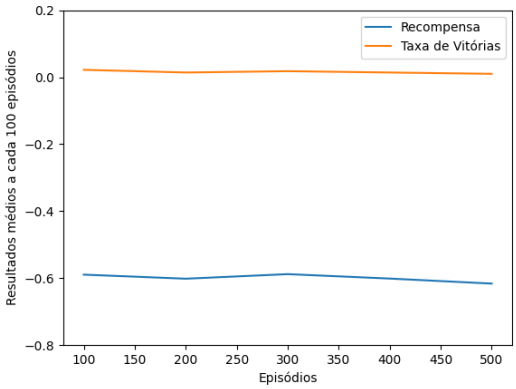


Figura 4.4 – Média dos testes da rede linear com melhor resultado.

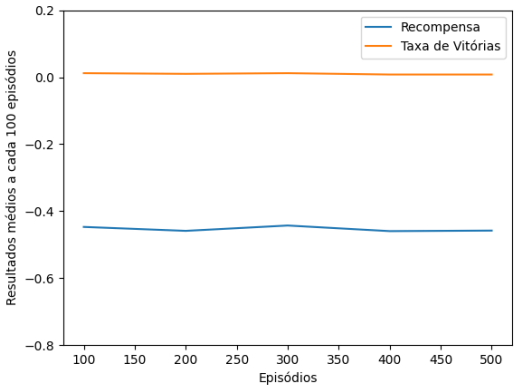


Figura 4.5 – Média dos testes da rede linear com pior resultado.

Como fica evidente a partir dos testes, em nenhum dos treinamentos a rede linear obteve bons resultados aproximando a função valor do problema. Ao final do treinamento, a taxa de vitórias foi ligeiramente melhor que a versão completamente aleatória. Um dos principais problemas dessa versão foi que a disposição dos dados de entrada em uma matriz com as mesmas dimensões que o mapa não facilitam o aprendizado da rede. Algumas soluções poderiam ser testadas para resolver esse problema, como um pré-processamento da entrada para extrair parâmetros mais específicos, como distâncias entre unidades, dano causado e sofrido caso de ataque, e utilizar estes parâmetros como entradas ao invés da matriz do mapa com os dados dispersos. Porém, para aproveitar a disposição espacial dos dados de entrada já obtidos, optou-se por uma abordagem não-linear com redes neurais convolucionais.

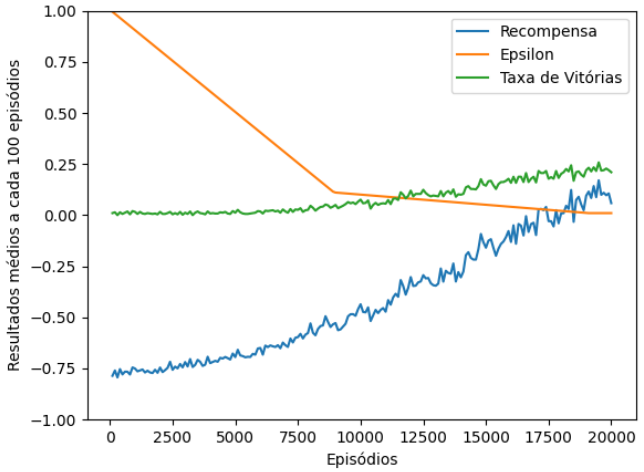
4.3 Versão Não-Linear

O processo de treinamento da versão não linear do agente consistiu na execução de 20.000 partidas. Para cada partida foi contabilizada a recompensa média e o resultado, derrota ou vitória. Na Figura 4.6 pode-se observar os resultados médios entres os cinco treinamentos. Nas Figuras 4.7 e 4.8, são representados, respectivamente, os treinamentos com o melhor e o pior resultados, considerando a taxa de vitórias nos últimos 500 episódios.

Como pode ser observado, a curva do gráfico com melhor resultado se aproxima da curva com os resultados médios. O pior caso, em contra-partida, começa a diminuir a recompensa média e a taxa de vitórias nos últimos 2000 episódios e foi uma exceção nos cinco treinamentos realizados. Um ponto a se destacar nos gráficos é que, como as recompensas de vitória e derrota são consideravelmente maiores que as das outras ações, quedas e aumentos na taxa de vitórias causam grande impacto também na recompensa média dos episódios.

Para avaliar melhor as redes treinadas foram utilizados ambas as redes resultantes do treinamento com melhor e pior resultado, também considerando taxa de vitórias e recompensa médias nos últimos 500 episódios. Nestes testes foi utilizado ϵ igual à zero e não foi feito nenhum novo treinamento durante a execução. Para cada rede, foram executados cinco testes, cada um composto por 500 partidas. Nas Figuras 4.9 e 4.10 são expostos os resultados médios dos testes da melhor e pior rede, respectivamente. É possível observar que, além da maior taxa de vitórias e recompensa média, a versão treinada com melhores resultados também possui maior consistência, obtendo resultados sem muita variação na maioria dos episódios.

Um dos principais fatores que pode ter contribuído para a diferença entre as duas redes é a aleatoriedade envolvida no processo de treinamento. Primeiramente, as ações no início do treinamento são tomadas de forma aleatória e as próprias ações do oponente



Padronizar as CORES!!! Por exemplo, a Taxa de Vitórias era LARANJA na outra Figura e VERDE nessa.

Figura 4.6 – Média dos resultados dos cinco treinamentos da versão não-linear.

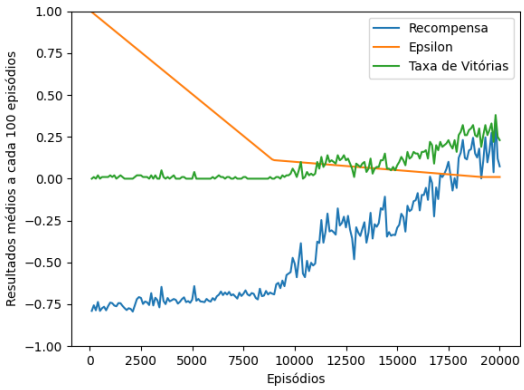


Figura 4.7 – Melhor resultado dos cinco treinamentos da versão não-linear.

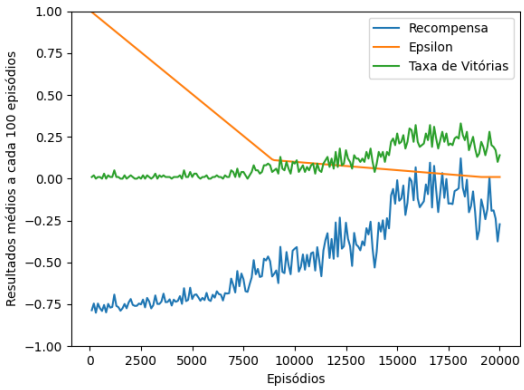


Figura 4.8 – Pior resultado dos cinco treinamentos da versão não-linear.

variam bastante. Assim, em cada treinamento a memória de repetição utilizada pode conter dados sobre partidas muito diferentes. O segundo fator importante é que a escolha das ações para o treinamento a partir da memória de repetição também é efetuada de maneira aleatória, então mesmo que seu conteúdo seja igual, o processo de treinamento ocorrerá de maneira diferente majoritariamente. A tendência é que quanto maior o número de episódios de treinamento e seções de treinamento realizadas a partir dos episódios salvos, menor será essa diferença. Dessa forma, utilizar um número maior de partidas e treinamentos pode diminuir a diferença entre os resultados de diferentes treinamentos.

Apesar desse problema de variação entre os resultados do treinamento, a versão não linear implementada apresentou resultados substancialmente melhores. O treinamento com melhores resultados obteve uma taxa de vitórias média de 32,97%, enquanto o com piores resultados conseguiu 24,88% de vitórias em média.

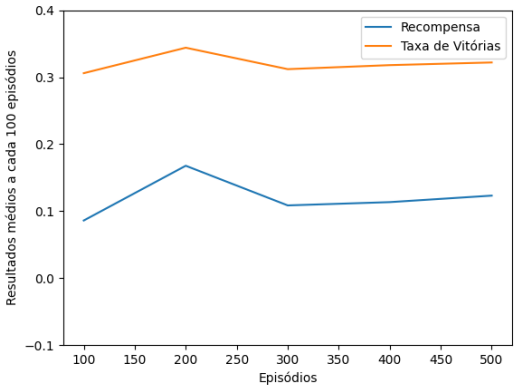


Figura 4.9 – Média dos testes da rede com melhor resultado.

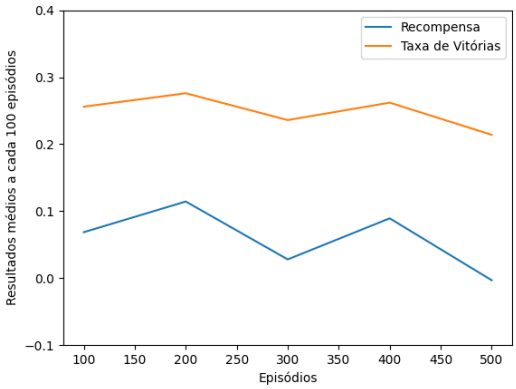


Figura 4.10 – Média dos testes da rede com pior resultado.

4.4 Comparação

Uma comparação entre os resultados das três implementações propostas pode ser observada na Tabela 4.1. Pode-se observar que ambas as implementações, tabular e com aproximação de funções lineares, apresentaram resultados ruins, apenas um pouco melhores que um comportamento completamente aleatório. Já a versão não-linear apresentou resultados consideravelmente melhores que as duas primeiras, vencendo uma em cada três partidas.

Tabela 4.1 – Taxa de Vitórias das Implementações

Versão do Treinamento	**Implementação**		
	Tabular	**Linear**	**Não-Linear**
Melhor Treinamento	1,85%	3,08%	32,97%
Pior Treinamento	1,10%	1,16%	24,88%

5. TRABALHOS RELACIONADOS

Diversas pesquisas foram feitas sobre inteligência artificial em jogos, muitas delas com o subgênero dos jogos de estratégia. Estas pesquisas utilizam tanto abordagens de IA clássica, como planejamento, quanto aplicações de aprendizado, como aprendizado por reforço ou por reforço profundo. A seguir serão apresentados alguns trabalhos relacionados com a implementação desenvolvida neste trabalho.

Amato e Shani utilizaram aprendizado por reforço para definir estratégias de alto nível no jogo *Civilization IV* [AS10]. A proposta dos autores consiste em controlar, através dos algoritmos de aprendizado Dyna-Q e *Q-learning*, a abordagem utilizada nas decisões do jogo, podendo ela ser focada em conquista militar, cultura, diplomacia, etc. As ações de baixo nível, como mover tropas e realizar ataques, são tratadas por algoritmos do estáticos. A proposta implementada se diferencia da apresentada por Amato e Shani pois as decisões efetuadas pelo agente são escolhas das ações de baixo nível, como movimentos, ataques e recrutamentos ao invés de ações de alto nível.

Santoso e Supriana utilizam uma combinação de algoritmo minimax e aprendizado por reforço na implementação de um agente para jogos de estratégia [SS14]. Os autores definem listas de ações baseadas no tipo da unidade e o agente escolhe uma delas a partir de seus pesos. Os pesos das ações tomadas são atualizados no final do turno considerando os resultados. A principal diferença entre a proposta dos dois autores e a apresentada neste trabalho está em como são tratadas as ações: Santoso e Supriana utilizam ações de nível um pouco mais alto, como atacar o inimigo mais próximo, longe ou fraco, dentre outras, enquanto a abordagem proposta utiliza ações de baixo nível, como se mover para uma coordenada x e y, ou atacar uma unidade em determinada coordenada. Por fim, o agente implementado utilizada apenas aprendizado por reforço, enquanto o dos autores também utiliza algoritmo minimax.

Leece e Jhala propõem o uso aprendizado por reforço na implementação de um agente com raciocínio espacial para o jogo *Planet Wars* [LJ13]. Em seu trabalho os autores aplicam *Q-learning* em um ambiente com diversos atores, os planetas. Para dar a cada ator informação espacial, eles atribuem a cada planeta um nível de ameaça, que indica aos planetas vizinhos a necessidade de apoio. A implementação proposta usa uma abordagem semelhante na versão tabular, utilizando distancias entre a unidade agindo, seus inimigos e construções para dar informações espaciais ao agente. Já na versão com aproximação de função, a entrada é uma matriz correspondente ao mapa que é utilizado pelas camadas convolucionais para informações espaciais, diferente dos autores que calculam as informações espaciais com um algoritmo estático para usá-las como entrada no agente.

Dans apresenta uma solução para o jogo *Battle for Wesnoth* utilizando algoritmos de planejamento [Dan16]. O autor divide a implementação em sub-tarefas como análise de

mapa, identificação de grupos de unidades, definição de instruções e controle individual de unidades. O trabalho de Dans difere do apresentado na escolha de abordagem para implementação do problema: enquanto ele utiliza algoritmos de planejamento, neste trabalho são apresentados algoritmos de aprendizado por reforço.

6. CONCLUSÃO

Este trabalho avaliou o uso de diferentes métodos de aprendizado por reforço em jogos de estratégia baseados em turnos, especificamente o jogo *Battle for Wesnoth*. Foram implementados agentes inicialmente com uma versão tabular, seguida por versões com aproximação de funções lineares e não lineares.

Na implementação tabular e na versão com aproximação de função linear não foram obtidos bons resultados. Na versão tabular muitas simplificações foram efetuadas para chegar em um número de estados e ações que pudesse ser treinado em um tempo razoável e, desse modo, o agente não conseguiu avaliar alguns aspectos do jogo corretamente. Já na versão linear, ~~foi utilizado~~ foi utilizada uma matriz contendo uma série de atributos para cada hexágono do mapa e a função linear utilizada não conseguiu aproximar a função valor de maneira apropriada.

A versão com aproximação de funções não lineares, no que lhe concerne, apresentou resultados promissores. A utilização de camadas convolucionais na rede neural artificial fez bom uso da distribuição espacial dos dados de entrada. Nos melhores resultados a rede apresentou uma taxa de vitória de cerca de 33%, que apesar de não ser ideal, mostra que a rede conseguiu aprender o comportamento necessário para ganhar pelo menos uma porção considerável das partidas.

Como melhorias futuras existem três principais aspectos a serem explorados: o treinamento, a arquitetura da rede e os dados de entrada. Devido às restrições de tempo para o desenvolvimento deste trabalho foi necessário limitar os treinamentos a períodos pequenos. Realizar treinamentos mais longos, explorando um número maior de partidas e mais seções de treinamento sobre os dados das partidas jogadas pode impactar em uma melhora dos resultados. Outra forma possível de melhorar a taxa de vitórias é testar arquiteturas mais elaboradas para a rede neural. Por fim, melhorar os dados de entrada, inserindo novas informações que possam ser úteis para a solução do problema pode melhorar o desempenho do agente.

REFERÊNCIAS BIBLIOGRÁFICAS

[ACT19] Arulkumaran, K.; Cully, A.; Togelius, J. “Alphastar: An evolutionary computation perspective”. In: Proceedings of the genetic and evolutionary computation conference companion, 2019, pp. 314–315.

[AS10] Amato, C.; Shani, G. “High-level reinforcement learning in strategy games.” In: AAMAS, 2010, pp. 75–82.

[BBC+19] Berner, C.; Brockman, G.; Chan, B.; Cheung, V.; Debiak, P.; Dennison, C.; Farhi, D.; Fischer, Q.; Hashme, S.; Hesse, C.; Józefowicz, R.; Gray, S.; Olsson, C.; Pachocki, J.; Petrov, M.; de Oliveira Pinto, H. P.; Raiman, J.; Salimans, T.; Schlatter, J.; Schneider, J.; Sidor, S.; Sutskever, I.; Tang, J.; Wolski, F.; Zhang, S. “Dota 2 with large scale deep reinforcement learning”, *CoRR*, vol. abs/1912.06680, 2019, 1912.06680.

[BCP+16] Brockman, G.; Cheung, V.; Pettersson, L.; Schneider, J.; Schulman, J.; Tang, J.; Zaremba, W. “Openai gym”, *arXiv preprint arXiv:1606.01540*, vol. 1–1, 2016.

[CHhH02] Campbell, M.; Hoane, A.; hsiung Hsu, F. “Deep blue”, *Artificial Intelligence*, vol. 134–1, 2002, pp. 57–83.

[Cyb89] Cybenko, G. V. “Approximation by superpositions of a sigmoidal function”, *Mathematics of Control, Signals and Systems*, vol. 2, 1989, pp. 303–314.

[Dan16] Dans, M. “Ia battle for wesnoth”, B.S. thesis, Universitat Politècnica de Catalunya, 2016, 52p.

[GBC16] Goodfellow, I.; Bengio, Y.; Courville, A. “Deep Learning”. MIT Press, 2016, 781p, http://www.deeplearningbook.org.

[LJ13] Leece, M. A.; Jhala, A. “Reinforcement learning for spatial reasoning in strategy games”. In: Ninth Artificial Intelligence and Interactive Digital Entertainment Conference, 2013, pp. 156–162.

[Mat21] Mattsc. “External ai control - the battle for wesnoth forums”. Capturado em: https://forums.wesnoth.org/viewtopic.php?f=10&t=51061, Nov 2021.

[MKS+15] Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; et al.. “Human-level control through deep reinforcement learning”, *nature*, vol. 518–7540, 2015, pp. 529–533.

[Nie15] Nielsen, M. A. “Neural networks and deep learning”. Determination press San Francisco, CA, 2015, vol. 25, 216p.

[Ope21] OpenAI. “How to create new environments for gym”. Capturado em: https://github.com/openai/gym/blob/master/docs/creating_environments.md, Nov 2021.

[RN10] Russell, S.; Norvig, P. “Artificial Intelligence: A Modern Approach”. Prentice Hall, 2010, 3 ed., 1132p.

[RP20] Risi, S.; Preuss, M. “From chess and atari to starcraft and beyond: How game ai is driving the world of ai”, *KI - Künstliche Intelligenz*, vol. 1–1, Feb 2020, pp. 7–17.

[SB18] Sutton, R. S.; Barto, A. G. “Reinforcement Learning: An Introduction”. Cambridge, MA: The MIT Press, 2018, 548p.

[SS14] Santoso, S.; Supriana, I. “Minimax guided reinforcement learning for turn-based strategy games”. In: 2014 2nd International Conference on Information and Communication Technology (ICoICT), 2014, pp. 217–220.

[Ste21] Stelter, D. “Benchmarking adversarial resilience learning using the example of ai development for battle for wesnoth”. Capturado em: https://github.com/DStelter94/ARLinBfW, Nov 2021.

[Wes21] Wesnoth. “Creating custom ais”. Capturado em: https://wiki.wesnoth.org/Creating_Custom_AIs, Nov 2021.