

UNIOESTE - UNIVERSIDADE ESTADUAL DO OESTE DO PARANÁ
CENTRO DE ENGENHARIAS E CIÊNCIAS EXATAS
CURSO DE CIÊNCIA DA COMPUTAÇÃO

Algoritmo neuro-evolutivo aplicado à mecânica de um jogo 2D

Filipe Souza Cavalcante

FOZ DO IGUAÇU

2017

Filipe Souza Cavalcante

Algoritmo neuro-evolutivo aplicado à mecânica de um jogo 2D

Monografia submetida à Universidade Estadual do Oeste do Paraná, Curso de Ciência da Computação - Campus de Foz do Iguaçu, como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Dr. Rômulo César Silva

FOZ DO IGUAÇU

2017

Filipe Souza Cavalcante

Algoritmo neuro-evolutivo aplicado à mecânica de um jogo 2D

Dr. Rômulo César Silva
Orientador(a)

Dr. Renato Bobsin Machado
Membro

Ma. Teresinha Arnauts Hachisuca
Membro

Dedico este trabalho ao meu amado pai e minha amada mãe, se há algo que faz diferença na formação da personalidade e na vida de uma pessoa é o amor que ela recebe. Vocês me educaram com amor, se dedicaram à minha educação como ser humano, me deram amor. Vocês fizeram de mim a pessoa que hoje sou, e eu só tenho motivos para agradecer.

Mais do que a educação formal que vocês me ofereceram e que sempre se esforçaram para que fosse a melhor, a formação humana foi o que de mais importante vocês fizeram por mim. Eu só posso retribuir tentando ser o melhor filho que pais como vocês merecem ter.

Sou e serei eternamente grato por tudo que vocês dedicaram a mim. Eu tenho muito orgulho de ser filho de vocês e muita admiração pelos pais que tenho. Obrigado por tudo.

Amo muito vocês!

Agradecimentos

Agradeço a minha família, pelo amor, carinho, paciência e ensinamentos, de forma especial ao meu pai Marcelo e à minha mãe Edineia, por não medirem esforços para que eu pudesse levar meus estudos adiante. Agradeço aos meus amigos, por confiarem em mim e estarem do meu lado em todos os momentos da vida. Agradeço ao meu professor Rômulo por toda orientação e ajuda que me foram dados. Agradeço esta instituição, pelo excelente ambiente oferecido aos seus alunos e os profissionais qualificados que disponibiliza para nos ensinar. Agradeço aos meus companheiros de trabalho no espaço de *coworking* na Incubadora Santos Dumont pelos bons momentos e risadas no dia a dia.

“Ciência da Computação está tão relacionada aos computadores quanto a Astronomia aos telescópios, Biologia aos microscópios, ou Química aos tubos de ensaio. A Ciência não estuda ferramentas. Ela estuda como nós as utilizamos, e o que descobrimos com elas.”

(Edsger Dijkstra)

Resumo

Machine learning ou aprendizado de máquina em inteligência artificial é um tema relacionado à implementação de programas de computador capazes de realizar o processo de aprendizagem de forma autônoma [HOSCH, 2009].

A neuro-evolução é uma técnica de aprendizado de máquina que aplica algoritmos evolucionários para construir uma rede neural artificial, tendo como inspiração o processo biológico evolutivo do sistema nervoso na natureza [LEHMAN; MIKKULAINEN, 2013].

Através da neuro-evolução é possível resolver não só problemas onde se tenham respostas conhecidas, como também desenvolver agentes inteligentes que aprendem a interagir com um determinado ambiente apropriadamente e encontrar soluções desconhecidas [ARAUJO; FRANCA, 2016].

Jogos eletrônicos são cenários que na sua grande maioria possuem problemas abstratos e de soluções desconhecidas, tornando-se um bom cenário para aplicação e avaliação de um algoritmo neuro-evolutivo.

O objetivo desta monografia é o estudo e desenvolvimento da aplicação de um algoritmo neuro-evolutivo a mecânica de um jogo eletrônico 2D, ou seja, um agente autônomo capaz de controlar a mecânica de um personagem e cumprir um devido objetivo. Obteve-se sucesso na implementação desse algoritmo.

A implementação completa do algoritmo pode ser encontrada e utilizada neste link da plataforma GitHub: <https://github.com/filipecavalc/Algoritmo-neuro-evolutivo-aplicado-a-mecanica-de-um-jogo-2D>.

Palavras-chaves: Inteligência Artificial, Neuro-evolução, Redes neurais, Algoritmo genético, Jogos eletrônicos, Python.

Listas de ilustrações

Figura 1 – Modelo genérico do processo de aprendizado de maquina.	3
Figura 2 – Hierarquia do aprendizado.	4
Figura 3 – Neurônio artificial.	8
Figura 4 – Neurônio biológico.	8
Figura 5 – Funcionamento do algoritmo genético.	9
Figura 6 – Representação da evolução de uma rede neural.	13
Figura 7 – Configurações iniciais do tabuleiro no othello.	18
Figura 8 – Gráfico da relação <i>fitness</i> e a evolução entre as gerações.	19
Figura 9 – Uma visão geral da arquitetura do agente.	20
Figura 10 – Arquitetura baseada em objeto.	21
Figura 11 – Arquitetura para pixels brutos.	21
Figura 12 – Arquitetura baseada em ruído semeado.	21
Figura 13 – Inimigos disponíveis para treino dos agentes autônomos.	25
Figura 14 – Possíveis combinações de controle para simulação.	26
Figura 15 – Possíveis configuração de objetivos.	27
Figura 16 – Sensores.	28
Figura 17 – Representação visual do algoritmo implementado.	31
Figura 18 – Inimigo e cenário utilizado para o agente especialista 1	41
Figura 19 – Evolução do <i>fitness</i> especialista inimigo 1 com população de tamanho 10.	42
Figura 20 – Evolução do <i>fitness</i> especialista inimigo 1 com população de tamanho 25.	43
Figura 21 – Evolução do <i>fitness</i> especialista inimigo 1 com população de tamanho 50.	43
Figura 22 – Evolução do <i>fitness</i> especialista inimigo 1 com população de tamanho 100.	44
Figura 23 – Evolução do <i>fitness</i> especialista inimigo 1 com população de tamanho 150.	44
Figura 24 – Variedade de espécies do <i>fitness</i> especialista inimigo 1 com população de tamanho 150.	45
Figura 25 – Inimigo e cenário utilizado para o agente especialista 2	45
Figura 26 – Evolução do <i>fitness</i> especialista inimigo 2 com população de tamanho 10.	46
Figura 27 – Evolução do <i>fitness</i> especialista inimigo 2 com população de tamanho 25.	46
Figura 28 – Evolução do <i>fitness</i> especialista inimigo 2 com população de tamanho 50.	47
Figura 29 – Evolução do <i>fitness</i> especialista inimigo 2 com população de tamanho 100.	47

Figura 30 – Evolução do <i>fitness</i> especialista inimigo 2 com população de tamanho 150.	48
Figura 31 – Variedade de espécies do <i>fitness</i> especialista inimigo 2 com população de tamanho 150.	48
Figura 32 – Inimigo e cenário utilizado para o agente especialista 3	49
Figura 33 – Evolução do <i>fitness</i> especialista inimigo 3 com população de tamanho 10.	50
Figura 34 – Evolução do <i>fitness</i> especialista inimigo 3 com população de tamanho 25.	50
Figura 35 – Evolução do <i>fitness</i> especialista inimigo 3 com população de tamanho 50.	51
Figura 36 – Evolução do <i>fitness</i> especialista inimigo 3 com população de tamanho 100.	51
Figura 37 – Variedade de espécies do <i>fitness</i> especialista inimigo 3 com população de tamanho 100.	52
Figura 38 – Evolução do <i>fitness</i> especialista inimigo 3 com população de tamanho 150.	52
Figura 39 – Variedade de espécies do <i>fitness</i> especialista inimigo 3 com população de tamanho 150.	53
Figura 40 – Inimigo e cenário utilizado para o agente especialista 4	53
Figura 41 – Evolução do <i>fitness</i> especialista inimigo 4 com população de tamanho 10.	54
Figura 42 – Evolução do <i>fitness</i> especialista inimigo 4 com população de tamanho 25.	54
Figura 43 – Evolução do <i>fitness</i> especialista inimigo 4 com população de tamanho 50.	55
Figura 44 – Evolução do <i>fitness</i> especialista inimigo 4 com população de tamanho 100.	55
Figura 45 – Variedade de espécies do <i>fitness</i> especialista inimigo 4 com população de tamanho 100.	56
Figura 46 – Evolução do <i>fitness</i> especialista inimigo 4 com população de tamanho 150.	56
Figura 47 – Variedade de espécies do <i>fitness</i> especialista inimigo 4 com população de tamanho 150.	57
Figura 48 – Inimigo e cenário utilizado para o agente especialista 5	57
Figura 49 – Evolução do <i>fitness</i> especialista inimigo 5 com população de tamanho 10.	58
Figura 50 – Evolução do <i>fitness</i> especialista inimigo 5 com população de tamanho 25.	58
Figura 51 – Evolução do <i>fitness</i> especialista inimigo 5 com população de tamanho 50.	59
Figura 52 – Evolução do <i>fitness</i> especialista inimigo 5 com população de tamanho 100.	59
Figura 53 – Variedade de espécies do <i>fitness</i> especialista inimigo 5 com população de tamanho 100.	60
Figura 54 – Evolução do <i>fitness</i> especialista inimigo 5 com população de tamanho 150.	60

Figura 55 – Variedade de espécies do <i>fitness</i> especialista inimigo 5 com população de tamanho 150.	61
Figura 56 – Inimigo e cenário utilizado para o agente especialista 6	61
Figura 57 – Evolução do <i>fitness</i> especialista inimigo 6 com população de tamanho 10.	62
Figura 58 – Evolução do <i>fitness</i> especialista inimigo 6 com população de tamanho 25.	62
Figura 59 – Evolução do <i>fitness</i> especialista inimigo 6 com população de tamanho 50.	63
Figura 60 – Evolução do <i>fitness</i> especialista inimigo 6 com população de tamanho 100.	63
Figura 61 – Evolução do <i>fitness</i> especialista inimigo 6 com população de tamanho 150.	64
Figura 62 – Variedade de espécies do <i>fitness</i> especialista inimigo 6 com população de tamanho 150.	64
Figura 63 – Inimigo e cenário utilizado para o agente especialista 7	65
Figura 64 – Evolução do <i>fitness</i> especialista inimigo 7 com população de tamanho 10.	66
Figura 65 – Evolução do <i>fitness</i> especialista inimigo 7 com população de tamanho 25.	66
Figura 66 – Evolução do <i>fitness</i> especialista inimigo 7 com população de tamanho 50.	67
Figura 67 – Evolução do <i>fitness</i> especialista inimigo 7 com população de tamanho 100.	67
Figura 68 – Variedade de espécies do <i>fitness</i> especialista inimigo 7 com população de tamanho 100.	68
Figura 69 – Evolução do <i>fitness</i> especialista inimigo 7 com população de tamanho 150.	68
Figura 70 – Variedade de espécies do <i>fitness</i> especialista inimigo 7 com população de tamanho 150.	69
Figura 71 – Inimigo e cenário utilizado para o agente especialista 8	69
Figura 72 – Evolução do <i>fitness</i> especialista inimigo 8 com população de tamanho 10.	70
Figura 73 – Evolução do <i>fitness</i> especialista inimigo 8 com população de tamanho 25.	70
Figura 74 – Evolução do <i>fitness</i> especialista inimigo 8 com população de tamanho 50.	71
Figura 75 – Evolução do <i>fitness</i> especialista inimigo 8 com população de tamanho 100.	71
Figura 76 – Variedade de espécies do <i>fitness</i> especialista inimigo 8 com população de tamanho 100.	72
Figura 77 – Evolução do <i>fitness</i> especialista inimigo 8 com população de tamanho 150.	72
Figura 78 – Variedade de espécies do <i>fitness</i> especialista inimigo 8 com população de tamanho 150.	73

Listas de Abreviaturas e Siglas

2D Duas dimensões

eager Método de aprendizagem ansiosa

lazy Método de aprendizagem preguiçosa

Machine learning Aprendizado de maquina

rtNEAT Real-Time Neuroevolution of augmenting topologies

CMA-ES *Covariance matrix adaptation evolution strategy*

HyperNEAT codificação de rede indireta

IA Inteligência artificial

NEAT *neuroevolution of augmenting topologies*

Sumário

1	Introdução	1
2	Aprendizado de máquina	2
2.1	Considerações iniciais	2
2.2	Conceito	2
2.3	Hierarquia do aprendizado	4
2.3.1	Aprendizado indutivo	4
2.3.1.1	Aprendizado supervisionado	4
2.3.1.2	Aprendizado não-supervisionado	5
2.3.1.3	Aprendizado semi-supervisionado	5
2.3.1.4	Aprendizado por reforço	6
2.4	Paradigmas de aprendizado	6
2.4.1	Simbólico	6
2.4.2	Estatístico	7
2.4.3	Baseado em exemplos	7
2.4.4	Conexionista	8
2.4.5	Genético	8
2.5	Considerações finais	9
3	Neuro-evolução	11
3.1	Considerações iniciais	11
3.2	Conceito	11
3.3	Porque neuro-evolução	12
3.4	Conceito básico do algoritmo	13
3.5	Métodos	14
3.6	Extensões	15
3.7	Aplicabilidade	17
3.7.1	Othello	18
3.7.2	MarI/O	19
3.7.3	<i>Starcraft: Brood War</i>	19
3.7.4	Uma abordagem geral para <i>game playing</i> no video game Atari	20
3.8	Considerações finais	21
4	Framework EvoMan	23
4.1	Considerações iniciais	23
4.2	Por que o framework EvoMan	23

4.3	Estrutura	24
4.4	Controles	24
4.5	Cenário e inimigos	24
4.6	Tipos de simulações	26
4.7	Objetivos de treinamento	27
4.8	Sensores	27
4.9	<i>Fitness</i>	28
4.10	Considerações finais	29
5	Implementação do algoritmo	30
5.1	Considerações iniciais	30
5.1.1	Linguagem	30
5.2	Algoritmo	31
5.2.1	Biblioteca <i>NEAT-Python</i>	32
5.2.2	Instância de ambiente com <i>framework</i> <i>EvoMan</i>	33
5.2.3	Estrutura de controle	34
5.2.4	Avaliação de genomas	35
5.2.5	Configuração da população de redes neurais	36
5.2.6	Execução	38
5.3	Testes e Resultados	41
5.3.1	Especialista inimigo 1	41
5.3.1.1	Resultados com a população de tamanho 10	42
5.3.1.2	Resultados com a população de tamanho 25	43
5.3.1.3	Resultados com a população de tamanho 50	43
5.3.1.4	Resultados com a população de tamanho 100	44
5.3.1.5	Resultados com a população de tamanho 150	44
5.3.2	Especialista inimigo 2	45
5.3.2.1	Resultados com a população de tamanho 10	46
5.3.2.2	Resultados com a população de tamanho 25	46
5.3.2.3	Resultados com a população de tamanho 50	47
5.3.2.4	Resultados com a população de tamanho 100	47
5.3.2.5	Resultados com a população de tamanho 150	48
5.3.3	Especialista inimigo 3	49
5.3.3.1	Resultados com a população de tamanho 10	50
5.3.3.2	Resultados com a população de tamanho 25	50
5.3.3.3	Resultados com a população de tamanho 50	51
5.3.3.4	Resultados com a população de tamanho 100	51
5.3.3.5	Resultados com a população de tamanho 150	52
5.3.4	Especialista inimigo 4	53

5.3.4.1	Resultados com a população de tamanho 10	54
5.3.4.2	Resultados com a população de tamanho 25	54
5.3.4.3	Resultados com a população de tamanho 50	55
5.3.4.4	Resultados com a população de tamanho 100	55
5.3.4.5	Resultados com a população de tamanho 150	56
5.3.5	Especialista inimigo 5	57
5.3.5.1	Resultados com a população de tamanho 10	58
5.3.5.2	Resultados com a população de tamanho 25	58
5.3.5.3	Resultados com a população de tamanho 50	59
5.3.5.4	Resultados com a população de tamanho 100	59
5.3.5.5	Resultados com a população de tamanho 150	60
5.3.6	Especialista inimigo 6	61
5.3.6.1	Resultados com a população de tamanho 10	62
5.3.6.2	Resultados com a população de tamanho 25	62
5.3.6.3	Resultados com a população de tamanho 50	63
5.3.6.4	Resultados com a população de tamanho 100	63
5.3.6.5	Resultados com a população de tamanho 150	64
5.3.7	Especialista inimigo 7	65
5.3.7.1	Resultados com a população de tamanho 10	66
5.3.7.2	Resultados com a população de tamanho 25	66
5.3.7.3	Resultados com a população de tamanho 50	67
5.3.7.4	Resultados com a população de tamanho 100	67
5.3.7.5	Resultados com a população de tamanho 150	68
5.3.8	Especialista inimigo 8	69
5.3.8.1	Resultados com a população de tamanho 10	70
5.3.8.2	Resultados com a população de tamanho 25	70
5.3.8.3	Resultados com a população de tamanho 50	71
5.3.8.4	Resultados com a população de tamanho 100	71
5.3.8.5	Resultados com a população de tamanho 150	72
5.4	Considerações finais	73
6	Conclusões e trabalhos futuros	74
Referências		75

1 Introdução

O primeiro jogo eletrônico documentado surgiu no ano de 1952, um simples jogo da velha e desde então a complexidade e realismo dos jogos eletrônicos supera qualquer expectativa da época em que esse meio de entretenimento surgiu [KCTS, 2015].

A tarefa de dominar a complexidade da mecânica de jogos eletrônicos 2D é um cenário adequado para testar diversas técnicas de inteligência artificial, incluindo técnicas de **Machine learning** (aprendizado de máquina), por sua capacidade de simular um ambiente dinâmico, ou seja, que não apresenta sempre o mesmo comportamento [MIRAS, 2016].

A neuro-evolução é uma técnica de aprendizado de máquina que aplica algoritmos evolucionários para construir uma rede neural artificial, tendo como inspiração o processo biológico evolutivo do sistema nervoso na natureza[LEHMAN; MIIKKULAINEN, 2013].

Através da neuro-evolução é possível resolver não só problemas onde se tenham respostas conhecidas, como também desenvolver agentes inteligentes que aprendem a interagir com um determinado ambiente apropriadamente e encontrar soluções desconhecidas [ARAUJO; FRANCA, 2016].

Desde que os algoritmos evolutivos foram propostos como método de otimização, eles têm sido usados com sucesso para resolver problemas complexos em diferentes áreas, por exemplo, automação do projeto de circuitos e equipamentos, planejamento de tarefas, mineração de dados, entre outros [ARAUJO; FRANCA, 2016].

Este trabalho compreende um estudo sobre a aplicação de técnicas de IA, mais especificamente de algoritmos neuro-evolutivos, para aprendizado autônomo da mecânica de jogos eletrônicos 2D.

Para realização da implementação e experimentos foram realizados pesquisas bibliográficas visando encontrar ferramentas de software adequadas ao propósito de geração do ambiente (cenário) em jogos eletrônicos 2D.

Esta monografia é composta de 6 capítulos: os capítulos 2 e 3 apresentam o referencial teórico usado no estudo. O capítulo 4 apresenta em detalhes o *framework* EvoMan[MIRAS, 2016], usado para geração dos cenários de teste. O capítulo 5 aborda o funcionamento da solução implementada, os resultados obtidos e uma breve discussão sobre os resultados. No capítulo 6 são apresentadas as conclusões e trabalhos futuros.

2 Aprendizado de máquina

2.1 Considerações iniciais

Em 1959, Arthur Samuel definiu aprendizado de máquina como o campo de estudo que dá a capacidade aos computadores de aprender sem serem explicitamente programados [SIMON, 2013].

Esses algoritmos funcionam construindo um modelo a partir de entradas e na interpretação dessas entradas que darão um sentido e possivelmente uma resposta a essas entradas.

O objetivo deste capítulo é apresentar os principais aspectos do aprendizado de máquina, conceitos sobre o tema para um melhor entendimento do seu contexto e função.

Os temas abordados são:

- O conceito de aprendizado de máquina para contextualizar o cenário em que se encaixa, explicando também o objetivo pelo qual a aprendizagem de máquina se fundamenta e como um modelo genérico do processo de aprendizado pode ser visualmente representado;
- As duas grandes classes de aprendizado de máquina;
- A hierarquia do aprendizado mostrando as diferentes formas aprendizado;
- Os paradigmas de aprendizado, sendo eles simbólico, estatístico, baseado em exemplos, conexionista e genético, sendo que cada um deles são a representação de formas de se realizar o processo de aprendizado;

2.2 Conceito

Machine learning ou aprendizado de máquina em inteligência artificial é um tema relacionado a implementação de programas de computador capazes de realizar o processo de aprendizagem de forma autônoma [HOSCH, 2009].

A utilização de algoritmos de aprendizado de máquina em tarefas de aquisição de conhecimento para solução de problemas é um trabalho que acontece através de “experiências vividas”, ou seja, através da experimentação e comparativos com diversos algoritmos com o objetivo de encontrar o que melhor se adapte à solução de um determinado problema [PRATI; BARANAUSKAS; MONARD, 2002].

Os algoritmos de aprendizado de máquina utilizam as experiências acumuladas através de soluções bem sucedidas de problemas anteriores para a tomada de decisões [MONARD; BARANAUSKAS, 2003].

A comparação deste tipo de algoritmo pode ser uma tarefa complicada, visto que cada um pode conter métricas de avaliação de resultados diferentes, tornando uma tarefa de complexidade relativamente alta [PRATI; BARANAUSKAS; MONARD, 2002].

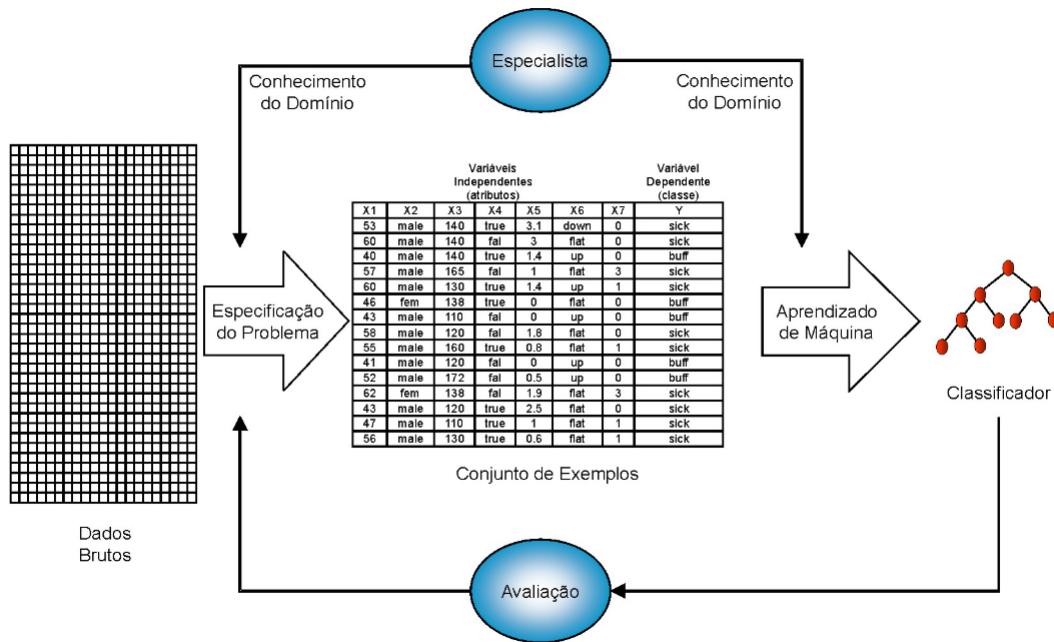


Figura 1: Modelo genérico do processo de aprendizado de máquina.

Fonte: Baranauskas [2007]

Existem várias formas de se escrever um algoritmo de aprendizagem de máquina. Essas diferentes formas possuem características particulares e similares entre elas, possibilitando a classificação quanto à linguagem de descrição, modo, paradigma e forma com que o algoritmo realiza o processo de aprendizado [MONARD; BARANAUSKAS, 2003].

Considera-se que um programa tem a capacidade de aprender quando consegue adquirir experiência com a execução de uma tarefa [HEINEN; OSÓRIO, 2005].

Segundo Monard e Baranauskas (2003 apud MICHALSKI; BRATKO; KUBAT, 1998) os diferentes tipos de sistemas de aprendizado podem ser enquadrados em duas grandes classes:

- Caixa-preta, onde o sistema desenvolve sua própria representação do conceito, isto é, sua representação interna não necessariamente poderá ser de fácil interpretação por humanos e tão pouco fornecem esclarecimento ou algum tipo de explicação sobre o processo de reconhecimento;

- Orientados a conhecimento que objetivam a criação de estruturas simbólicas que sejam compreensíveis por humanos.

2.3 Hierarquia do aprendizado

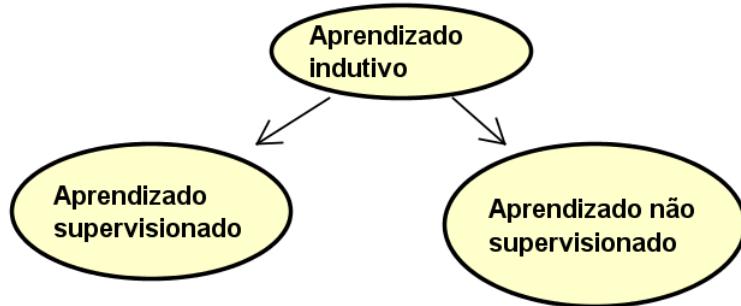


Figura 2: Hierarquia do aprendizado.

Fonte: Baranauskas [2007]

2.3.1 Aprendizado indutivo

A indução é um método de inferência lógica que permite chegar em conclusões genéricas sobre um conjunto específico de informações. A indução possui caracterização de um raciocínio, que tem origem a partir de um conceito específico, mas o generaliza dando significado ao todo. Na indução, apresenta-se um conceito que é aprendido realizando a inferência indutiva sobre as informações previamente apresentadas, fazendo com que as hipóteses geradas através desse método possam ou não preservar a verdade [MONARD; BARANAUSKAS, 2003].

A indução é o recurso mais utilizado pelo cérebro humano para aprender e gerar novos conhecimentos, mas ela deve ser utilizada com cautela, pois se o número de informações for insuficiente, ou se as informações não forem bem escolhidas, as hipóteses obtidas terão pouco valor [MONARD; BARANAUSKAS, 2003].

O aprendizado indutivo é dividido em supervisionado e não-supervisionado [BARANAUSKAS, 2007].

2.3.1.1 Aprendizado supervisionado

Segundo SAS Institute Inc [2015], aprendizado supervisionado são algoritmos que são treinados com informações rotuladas, onde a entrada tem uma saída conhecida. Por

exemplo:

- uma máquina poderia ter uma série de informações rotuladas como “FALHA” ou “SUCESSO”;
- O algoritmo de aprendizagem recebe um conjunto de entradas juntamente com as correspondentes saídas corretas, e o algoritmo aprende comparando as próprias saídas com as saídas corretas fornecidas;
- Em seguida, ele modifica o modelo de acordo com o resultado das comparações;
- Através de métodos de classificação e regressão, o aprendizado supervisionado consegue determinar padrões para prever os valores das saídas de informações não rotuladas.

As aplicações mais comuns que utilizam aprendizado supervisionado, são as que possuem uma série de dados históricos, que serão usados para prever ações futuras. Um exemplo seria o histórico de transações de um cartão de crédito, que irá determinar o padrão de consumo do usuário do cartão. Dessa forma, pode-se prever fraudes de compras fora do padrão do proprietário do cartão de crédito [SAS INSTITUTE INC, 2015].

2.3.1.2 Aprendizado não-supervisionado

O aprendizado não-supervisionado é usado com informações não rotuladas. O algoritmo não recebe informação sobre a saída estar correta ou incorreta. O algoritmo precisa descobrir por conta própria, se a saída está apropriada para a entrada. O objetivo é explorar os dados e encontrar alguma forma de estruturá-los [SAS INSTITUTE INC, 2015].

O aprendizado não-supervisionado funciona muito bem com dados transacionais. Por exemplo, pode identificar os segmentos de clientes com atributos semelhantes que podem ser tratados de forma semelhante em campanhas de marketing. Ou pode encontrar os principais atributos que separam segmentos de clientes e diferenciá-los uns dos outros [SAS INSTITUTE INC, 2015].

As técnicas mais populares incluem mapas auto-organizados, mapeamento do vizinho mais próximo, agrupamento *k-means* e decomposição de valor singular. Esses algoritmos também são usados para segmentar tópicos de texto, recomendar itens e identificar valores-limite de dados [SAS INSTITUTE INC, 2015].

2.3.1.3 Aprendizado semi-supervisionado

É usado para aplicações do mesmo tipo que as para o aprendizado supervisionado. Utiliza tanto dados rotulados quanto os não rotulados para o treinamento, normalmente

com uma pequena quantidade de dados rotulados com uma grande quantidade de dados não rotulados, já que dados não rotulados tem custo menor e irão precisar de menos esforço para serem adquiridos [SAS INSTITUTE INC, 2015].

Esse tipo de aprendizagem pode ser usado com métodos como a classificação, regressão e previsão. O aprendizado semi-supervisionado tem utilidade quando o custo para rotular os dados é muito alto para permitir um processo de treinamento totalmente rotulado [SAS INSTITUTE INC, 2015].

2.3.1.4 Aprendizado por reforço

O aprendizado por reforço é de uso comum em aplicações desenvolvidas para a robótica, jogos e navegação. O aprendizado por reforço é um algoritmo que descobre pela tentativa e erro, quais ações geram os melhores resultados [SAS INSTITUTE INC, 2015].

Existem três componentes principais:

- O agente (quem toma as decisões)
- O ambiente (tudo que tem interação com o agente)
- As ações (todas as ações possíveis que um agente pode tomar)

O objetivo é que o agente escolha ações que geram os melhores resultados ao longo do tempo, assim otimizando cada vez mais as tomadas de decisão. Boas políticas para determinação da qualidade das escolhas levam a uma aprendizagem mais rápida [SAS INSTITUTE INC, 2015].

2.4 Paradigmas de aprendizado

Existem alguns paradigmas de aprendizado, tais como: Simbólico, Estatístico, Baseado em exemplos, Conexionista e Genético [BARANAUSKAS, 2007].

2.4.1 Simbólico

É um sistema de aprendizado que busca aprender através da criação de representações simbólicas de um conceito, realizando a análise de exemplos e contra-exemplos desse conceito [MONARD; BARANAUSKAS, 2003].

Esse sistema de aprendizado tem sua representação simbólica na forma de alguma expressão lógica, árvore de decisão ou rede semântica [MONARD; BARANAUSKAS, 2003].

2.4.2 Estatístico

O conceito geral no paradigma estatístico é o de utilizar modelos estatísticos para encontrar uma boa aproximação do conceito induzido [MONARD; BARANAUSKAS, 2003].

A maior parte desses métodos são paramétricos, assumem alguma forma de modelo, e encontram valores apropriados para os parâmetros do modelo a partir dos dados. Alguns autores consideram redes neurais como métodos estatísticos paramétricos, já que treinar uma rede neural tem como significado encontrar valores apropriados para pesos e bias pré-determinados [MONARD; BARANAUSKAS, 2003].

Entre os métodos estatísticos, segundo Monard e Baranauskas [2003], o aprendizado Bayesiano é o que se destaca. O aprendizado Bayesiano tem os seguintes preceitos:

- Utiliza um modelo probabilístico;
- É baseado no conhecimento prévio do problema;
- Combina tudo isso com exemplos de treinamento para determinar a probabilidade final de uma hipótese.

2.4.3 Baseado em exemplos

O paradigma baseado em exemplos funciona através da classificação dos exemplos por aproximação e similaridades entre eles, logo para os novos exemplos, a classificação será com base no aprendizado dos exemplos classificados anteriormente ao exemplo que está passando pelo processo de classificação [MONARD; BARANAUSKAS, 2003].

Este tipo de sistema de aprendizado é chamado de **lazy**. O sistema do tipo **lazy** precisa manter exemplos em memória para classificar novos exemplos, em oposição aos sistemas **eager**, que usam os exemplos para induzir o modelo, e após isso descarta os exemplos [AHA, 1997; MONARD; BARANAUSKAS, 2003].

O fato desse tipo de sistema ser do tipo **lazy** faz com que a escolha dos exemplos que serão mantidos em memória, um fator muito importante. A escolha desses exemplos deve ser composta pelos exemplos mais representativos do problema [MONARD; BARANAUSKAS, 2003].

Segundo Monard e Baranauskas [2003] no artigo *Instance-based learning algorithms* escrito por Aha, Kibler e Albert [1991] é descrito algumas estratégias para realizar a decisão de quando um novo exemplo deve ser memorizado.

As estratégias de vizinho mais próximo e raciocínio baseado em casos são, provavelmente os algoritmos mais conhecidos neste paradigma [MONARD; BARANAUSKAS, 2003].

2.4.4 Conexionista

Conexionismo é utilizado para descrever a área de estudo que envolve a representação de uma rede neural, que envolve unidades altamente interconectadas [MONARD; BARANAUSKAS, 2003].

Uma rede neural é a construção matemática simplificada inspirada no modelo biológico do sistema nervoso [MONARD; BARANAUSKAS, 2003].

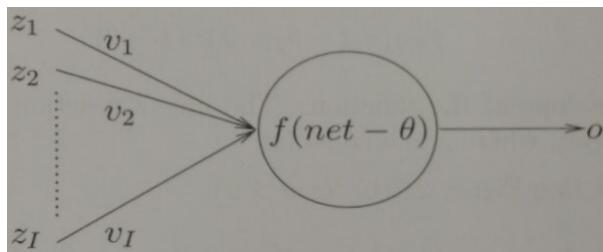


Figura 3: Neurônio artificial.

Fonte: Engelbrecht [2007]

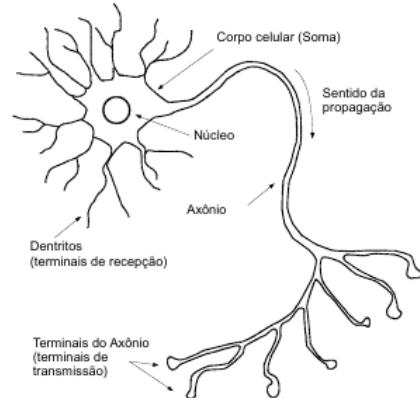


Figura 4: Neurônio biológico.

Fonte: Rodrigues [2009]

A ligação biológica com as conexões neurais do sistema nervoso tem despertado o interesse de pesquisadores, e tem levantado inúmeras discussões sobre os méritos e as limitações desse tipo de abordagem de aprendizado [MONARD; BARANAUSKAS, 2003].

Na sua grande maioria, as analogias com a Biologia têm levado muitos pesquisadores a acreditar que as redes neurais possuem um grande potencial em resoluções de problemas que requerem intensas formas de processamento sensorial humano, assim como visão e reconhecimento de voz [MONARD; BARANAUSKAS, 2003].

2.4.5 Genético

Segundo Monard e Baranauskas (2003 apud GOLDBERG, 1989), este paradigma de aprendizado é derivado do modelo evolucionário de aprendizado. Um classificador genético consiste:

- De uma população de elementos de classificação que competem para fazer a predição;

- Elementos que possuem uma performance fraca são descartados;
- O atributo que “quantifica” a performance geralmente é chamado de *fitness*;
- Enquanto isso os elementos mais fortes proliferam, produzindo variações de si mesmos.

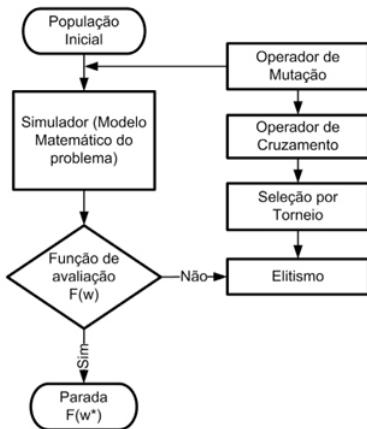


Figura 5: Funcionamento do algoritmo genético.

Fonte: Calixto et al. [2014]

Este paradigma possui uma analogia direta com a Teoria de Darwin de seleção natural, na qual sobrevivem os mais bem adaptados ao ambiente [MONARD; BARANAUSKAS, 2003].

2.5 Considerações finais

Este capítulo teve como objetivo o estudo dos conceitos e temas relacionados ao aprendizado de máquina para um melhor entendimento sobre não somente do tema em si, mas também da importância, funcionamento e comportamento.

Como resultado do conhecimento adquirido por este estudo, está determinado o uso de qual ramo da hierarquia do aprendizado, no caso o aprendizado não-supervisionado. Outra decisão vinda do resultado do estudo deste capítulo foi o uso de redes neurais e algoritmos genéticos para o processo de aprendizado do algoritmo que será escrito.

O entendimento básico da estrutura de um neurônio e a lógica básica por trás da rede neural e o processo evolucionário do algoritmo genético também será importante para uma implementação mais sólida do algoritmo.

No capítulo 3 será abordado com maior profundidade como os temas redes neurais e algoritmos genéticos se relacionam para funcionarem em conjunto e gerar a técnica que

é conhecido como neuro-evolução, assim como as diferentes formas de aplicação dessa metodologia e detalhes mais técnicos do algoritmo.

3 Neuro-evolução

3.1 Considerações iniciais

A neuro-evolução é uma forma de inteligência artificial que utiliza algoritmos evolucionários para gerar os parâmetros de uma rede neural, topologia e regras [STANLEY, 2017].

O objetivo desse capítulo é abordar o tema principal desta monografia com um maior grau de aprofundamento e detalhes desta técnica de inteligência artificial, visando compreender o que é necessário para a implementação de um algoritmo neuro-evolutivo.

Os tópicos abordados neste capítulo são:

- Conceito de neuro-evolução, explicando sucintamente o que é;
- As justificativas teóricas para a escolha do autor em utilizar neuro-evolução para uma criar um agente autônomo em um jogo eletrônico;
- Os conceitos básicos do algoritmo neuro-evolutivo, incluindo detalhamento passo a passo do algoritmo;
- Os diferentes métodos neuro-evolutivos que podem ser implementados;
- Quais as extensões da neuro-evolução desde como ser mais eficiente e também as diferentes propriedades que esse tipo de técnica pode ter;
- E por fim a aplicabilidade de um algoritmo neuro-evolutivo, incluindo em quais campos podem ser aplicados e uma série de exemplos de aplicações em jogos eletrônicos de vários tipos e gêneros diferentes, mostrando a flexibilidade que o método pode ter.

3.2 Conceito

Desde que os algoritmos evolutivos foram propostos como método de otimização, eles têm sido usados com sucesso para resolver problemas complexos em diferentes áreas, por exemplo, automação do projeto de circuitos e equipamentos, planejamento de tarefas, mineração de dados, entre outros [ARAUJO; FRANCA, 2016].

O sucesso de algoritmos evolutivos se deve, entre outras coisas, ao fato desse tipo de algoritmo em geral não precisar de formulações matemáticas complexas a respeito do

problema que se deseja resolver, além de oferecer alto grau de paralelismo no processo de busca [CRUZ, 2007].

Os algoritmos evolutivos embora vantajosos, em alguns casos podem acarretar em um alto custo computacional para avaliação das soluções durante o processo de busca, tornando a otimização por algoritmos evolutivos lenta para situações em que se deseja uma resposta rápida. Existem algumas técnicas para superar essas situações [CRUZ, 2007].

A neuro-evolução é uma técnica de aprendizado de máquina que aplica algoritmos evolucionários para construir uma rede neural artificial, tendo como inspiração o processo biológico evolutivo do sistema nervoso na natureza[LEHMAN; MIIKKULAINEN, 2013].

Comparado a outros métodos de aprendizagem de redes neurais, a neuro-evolução é altamente genérica, isso permite o aprendizado sem um objetivo explícito a ser atingido, com intervalos grandes entre *feedbacks* e com modelos de neurônios e estruturas da rede de neurônios arbitrários [LEHMAN; MIIKKULAINEN, 2013].

A neuro-evolução é uma aproximação efetiva para resolver problemas de aprendizagem por reforço e também é o método mais comum aplicado na robótica evolucionária e em vida artificial [LEHMAN; MIIKKULAINEN, 2013].

3.3 Porque neuro-evolução

A neuro-evolução é frequentemente aplicada a jogos eletrônicos. Este tipo de algoritmo trata a busca pela solução de um problema como uma tarefa de otimização, onde podemos definir uma medida de sucesso, chamada *fitness*, para ser maximizada ou minimizada tentando alcançar o melhor resultado possível em um tempo experimental a se determinar [ARAUJO; FRANCA, 2016].

A aplicação da neuro-evolução pode ser realizada de duas formas: um modelo simples, que evolui somente os pesos das conexões da rede, ou de forma completa em que se pode evoluir completamente a rede neural incluindo neurônios e conexões [ARAUJO; FRANCA, 2016]. O método utilizado foi a evolução completa.

Através da neuro-evolução é possível resolver não só problemas onde se tinham respostas conhecidas, como também desenvolver agentes inteligentes que aprendem a interagir com um determinado ambiente apropriadamente e encontrar soluções desconhecidas [ARAUJO; FRANCA, 2016].

Jogos eletrônicos são cenários que na sua grande maioria possuem problemas abstratos e de soluções desconhecidas, tornando-se um bom cenário para aplicação e avaliação de um algoritmo neuro-evolutivo.

3.4 Conceito básico do algoritmo

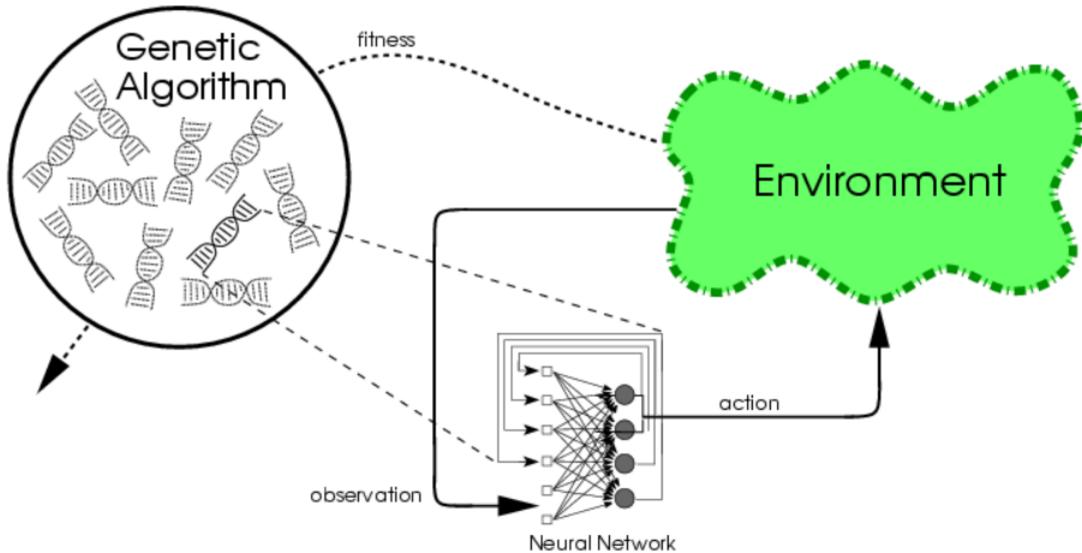


Figura 6: Representação da evolução de uma rede neural.

Fonte: Lehman e Miikkulainen [2013]

Normalmente na neuro-evolução, a população de codificações genéticas da rede neural é evoluída com o objetivo de encontrar uma rede que resolva uma dada tarefa [LEHMAN; MIIKKULAINEN, 2013].

A maior parte dos métodos neuro-evolutivos segue o ciclo repetitivo de geração e teste de algoritmos evolutivos da Figura 6. Segundo Lehman e Miikkulainen [2013] esse ciclo repetitivo de teste, avaliação dos resultados dos testes e aplicação do algoritmo genético nas redes neurais segue a seguinte sequência de passos:

- Cada codificação de uma população (um genótipo) é escolhida e por sua vez é descodificada em uma rede neural (um fenótipo) correspondente;
- Essa rede por sua vez é empregada na tarefa, e seu desempenho ao longo do tempo é medido, obtendo assim o valor de aptidão (*fitness*) para o genótipo correspondente;
- Após todos os membros da população serem avaliados seguindo este mesmo método, os operadores genéticos são usados para criar a próxima geração da população;
- As codificações com o resultado de aptidão mais altos passam por um processo aleatório de mutação e são cruzados uns com os outros, e a descendência resultante substitui os genótipos com menor aptidão na população;

- Logo o processo consiste em uma busca paralela inteligente em direção aos melhores genótipos, e continua até uma rede com uma condição de aptidão suficientemente alta para atender a tarefa com o nível de eficiência procurado.

3.5 Métodos

Segundo Lehman e Miikkulainen [2013], existem vários métodos para evolução das redes neurais, dependendo de como as redes estão codificadas:

- A codificação mais direta, chamada de neuro-evolução convencional ou simples, é formada pela concatenação dos valores numéricos para os pesos da rede. Essa codificação permite que o processo evolutivo do algoritmo, otimize os pesos de uma arquitetura de rede neural fixa, ou seja, de topologia não variável, tornando-se uma técnica prática e fácil de ser implementada em vários problemas de domínios diferentes. No entanto:
 - A escolha de um tamanho ou estrutura de rede neural pode levar a uma otimização com nível de *fitness* não adequado ao desejado;
 - Requer que seja feita experimentação de diferentes topologias da rede neural para melhor adequar-se a solução desejada;
 - Também possuem uma escalabilidade fraca para problemas de maior complexidade e dificuldade por conta dos parâmetros envolvidos que crescem linearmente ou quadraticamente em relação ao tamanho da rede neural.
- A partir dos problemas da neuro-evolução simples, surgiram soluções mais sofisticadas para resolvê-los. Uma das soluções se dá ao considerar o processo de evolução a um nível mais baixo, dividindo a rede neural em várias partes, ou seja, ao invés de possuir uma população de redes neurais completas, tem-se uma população de fragmentos de redes neurais, neurônios e pesos de conexão evoluídos. Cada fragmento é avaliado como uma parte de uma rede completa, e sua adequação reflete o quanto bem ele coopera com outros indivíduos nas soluções específicas que ele ajuda a compor. Desse modo, o problema complexo de encontrar uma rede neural que solucione o problema é dividido consideravelmente em subproblemas;
- Outra estratégia é evoluir a topologia da rede juntamente com os pesos da rede neural. A motivação dessa estratégia é resolver justamente o problema de definição da topologia de uma rede neural que seja adequada a solução do problema, substituindo o método ineficiente de experimentação de tentativa e erro até chegar em uma topologia de rede neural adequada. Uma vantagem adicional é a de que evoluindo a topologia consegue-se alcançar um melhor desempenho da rede neural do

que somente evoluindo os pesos. Uma abordagem de evolução promissora para este método é começar de forma gradual com problemas simples e ir progredindo para problemas mais complexos. Este tipo de método ocorre na biologia e é uma abordagem muito poderosa no aprendizado de máquina para aplicações em geral;

- Todos os métodos descritos acima tem como base o mapeamento da codificação genética de forma direta para uma rede neural correspondente: cada parte codificada corresponde uma parte da rede e vice-versa. A codificação indireta, em contrapartida, especifica um processo através do qual a rede é construída, como divisão celular, incorporação espacial ou geração através da expansão de regras gramaticais. Essa codificação pode ser altamente compacta e também aproveitar os benefícios das soluções modulares. A modularidade implica com que as mesmas estruturas tenham a capacidade de serem replicadas com pequenas modificações, como muitas vezes ocorre na biologia. A escalação próspera da neuro-evolução em níveis de complexidade biológica provavelmente depende do desenvolvimento de codificações indiretas mais sofisticadas, tornando essas codificações uma direção importante para futuras pesquisas [LEHMAN; MIIKKULAINEN, 2013].

3.6 Extensões

Os mecanismos básicos da neuro-evolução podem ser aumentadas de várias maneiras, tornando o processo mais eficiente e facilitando a resolução de problemas mais difíceis. Uma das extensões mais básicas é a evolução incremental, ou a modelagem: a evolução começa em uma tarefa simples, e uma vez que é dominada, as soluções são desenvolvidas ainda mais em uma tarefa mais desafiadora, e através de uma série dessas etapas de transferência, eventualmente na realidade tarefa de objetivo em si. A modelagem pode funcionar através da alteração do ambiente, como aumentando gradualmente a dificuldade da tarefa, ou alterando a função de aptidão física, recompensando comportamentos cada vez mais complexos. Muitas vezes, é possível resolver tarefas desafiadoras abordando-as de forma incremental mesmo quando não possam ser resolvidas diretamente [LEHMAN; MIIKKULAINEN, 2013].

Muitas extensões gerais para métodos de computação evolutiva aplicam-se particularmente bem à neuro-evolução. Por exemplo, técnicas de mutação inteligente, como as empregadas em estratégias evolutivas, são efetivas porque os pesos geralmente possuem correlações adequadas. Da mesma forma, os algoritmos evolutivos multiobjetivos que equilibram o desempenho em relação aos objetivos concorrentes podem muitas vezes beneficiar a neuro-evolução. As redes neurais também podem ser desenvolvidas através da coevolução, onde os indivíduos da população competem ou cooperam entre si. Em particular, uma corrida armamentista co-evolutiva pode ser facilitada por uma evolução

neurológica evolutiva de topologia: à medida que a rede se torna cada vez mais complexa, é provável que a evolução elabore os comportamentos existentes em vez de substituí-los. Finalmente, vários métodos de computação evolutiva geral para aumentar a exploração ou a diversidade dentro da população são muitas vezes efetivos e necessários ao aplicar a neuro-evolução em problemas difíceis. Por exemplo, reduzir a concorrência entre redes com diferentes topologias, idades ou funcionalidades muitas vezes melhora o desempenho [LEHMAN; MIIKKULAINEN, 2013].

Por outro lado, várias extensões utilizam as propriedades especiais do fenótipo da rede neural. Por exemplo, as funções de ativação de neurônios, estados iniciais e regras de aprendizado podem ser desenvolvidas para atender a uma determinada tarefa. Importante, a evolução pode ser combinada com outros métodos tradicionais de aprendizagem da rede neural (auto-organização, Perceptron de aprendizado supervisionado, etc). Em tais abordagens, enquanto a evolução geralmente fornece a rede inicial, ela se adapta mais durante sua avaliação na tarefa. A adaptação pode ocorrer através da plasticidade hebbiana (eficiência sináptica surge da estimulação repetida e persistente [ROQUE, 2015]) ou neuro-modulada, permitindo a adaptação através de *feedback* ambiental. Alternativamente, a aprendizagem supervisionada, como a propagação posterior pode ser usada, desde que os alvos estejam disponíveis. Curiosamente, mesmo que os comportamentos ótimos não sejam conhecidos, esse treinamento pode ser útil: as redes podem ser treinadas para imitar os indivíduos mais bem-sucedidos da população, ou parte da rede pode ser treinada para prever os próximos insumos. As mudanças de peso podem ser codificadas de volta ao genótipo, implementando a evolução Lamarckiana (Lamarck acreditava que mudanças no ambiente causavam mudanças nas necessidades dos organismos que ali viviam, causando mudanças no seu comportamento [LAMARCK, 1803]); alternativamente, eles podem afetar a seleção através do efeito Baldwin, ou seja, as redes que aprendem bem serão selecionadas para reprodução, mesmo que as mudanças de peso não sejam herdadas [LEHMAN; MIIKKULAINEN, 2013].

Existem também várias maneiras de polarizar e direcionar o sistema de aprendizagem usando o conhecimento humano. Por exemplo, as regras codificadas por humanos podem ser codificadas em estruturas de rede parciais e incorporadas nas redes em desenvolvimento como mutações estruturais. Esse conhecimento pode ser usado para implementar comportamentos iniciais na população, ou pode servir de conselho durante a evolução. Nos casos em que o conhecimento baseado em regras não está disponível, ainda pode ser possível obter exemplos de comportamento humano. Tais exemplos podem então ser incorporados na evolução, seja como componentes da aptidão, ou por treinamento explícito das soluções evoluídas para o comportamento humano, por exemplo através da transposição. Da mesma forma que foi discutido acima, o conhecimento sobre a tarefa e seus componentes pode ser utilizado na concepção de estratégias efetivas de modelagem.

Dessa forma, a experiência humana pode ser usada para inicializar e orientar a evolução em tarefas difíceis, bem como direcioná-la para os tipos desejados de soluções [LEHMAN; MIIKKULAINEN, 2013].

3.7 Aplicabilidade

Os métodos de neuro-evolução são poderosos especialmente em domínios contínuos de aprendizagem de reforço e aqueles que têm estados parcialmente observáveis. Esses domínios incluem muitas aplicações do mundo real de aprendizagem de reforço; a aplicação mais óbvia é o controle adaptativo e não-linear de dispositivos físicos. Por exemplo, os controladores de rede neural foram desenvolvidos para dirigir robôs móveis, automóveis e até foguetes. A abordagem de controle foi estendida para otimizar sistemas, como processos químicos, sistemas de fabricação e sistemas informáticos. No entanto, uma limitação crucial com as abordagens atuais é que os controladores geralmente precisam ser desenvolvidos em simulação e depois transferidos para o sistema real. A evolução é geralmente mais forte como um método de aprendizagem *off-line*, onde é livre para explorar soluções em potencial em paralelo [LEHMAN; MIIKKULAINEN, 2013].

A neuro-evolução provou ser útil na concepção de jogadores para jogos de tabuleiro, como damas, xadrez e othello. Curiosamente, a mesma abordagem funciona na construção de personagens em ambientes artificiais, como jogos e realidade virtual. Os personagens não jogadores nos jogos de vídeo atuais geralmente são roteirizados e limitados; A neuro-evolução pode ser usada para desenvolver comportamentos complexos para eles, e até mesmo adaptá-los em tempo real. Neuro-evolução pode assim facilitar novos tipos de videogames, como jogos onde os jogadores treinam uma equipe de agentes de IA , ou seja, o jogo se portando como o jogador aprendendo com ele e proporcionando um aumento de dificuldade no jogo proporcional ao avanço das habilidades do jogador. Da mesma forma, artefatos, como armas, podem ser construídos por redes neurais desenvolvidas, permitindo jogos onde os jogadores criam colaborativamente novos conteúdos no jogo que de outra forma deveriam ser explicitamente projetados por especialistas humanos [LEHMAN; MIIKKULAINEN, 2013].

Em terceiro lugar, a evolução das redes neurais é uma ferramenta natural para problemas na vida artificial e está sendo cada vez mais aplicada para explorar questões que são difíceis de investigar através de técnicas mais tradicionais em biologia evolutiva. Embora as pressões particulares de seleção que levaram a transições evolutivas fundamentais na natureza sejam transitórias e deixem pouca evidência direta, a neuro-evolução pode ser aplicada em experimentos controlados para investigar quais condições são necessárias para que certos comportamentos evoluam. Assim, é possível projetar experimentos de neuro-evolução sobre como comportamentos como busca, perseguição e evasão, caça e pastoreio,

colaboração e até mesmo a comunicação pode emergir em resposta à pressão ambiental. Neuro-evolução também pode ser aplicado para investigar tendências evolutivas mais abstratas, como a evolução da modularidade ou como o desenvolvimento biológico interage com a evolução. Além disso, a análise de circuitos neurais evoluídos e a compreensão de como eles se mapeiam para funcionar, podem levar a informações sobre as redes biológicas [LEHMAN; MIIKKULAINEN, 2013].

3.7.1 Othello

Othello é um jogo de tabuleiro 8x8 para dois jogadores. Um jogador possui peças brancas e outro peças pretas, todos discos idênticos, com uma face branca e outra preta [ANDERSEN; STANLEY; MIIKKULAINEN, 2002].

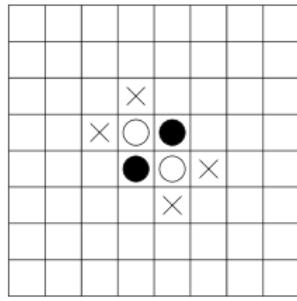


Figura 7: Configurações iniciais do tabuleiro no othello.

Fonte: Andersen, Stanley e Miikkulainen [2002]

Foi testado uma série de diferentes técnicas que foram sugeridas ao autor incluindo o algoritmo de busca alpha-beta, *temporal difference learning*, algoritmos genéticos, coevolução e NEAT [ANDERSEN; STANLEY; MIIKKULAINEN, 2002].

Evolução e coevolução foram usadas para desenvolver uma rede neural capaz de derrotar um jogador com o algoritmo alfa-beta. Embora a evolução tenha superado a coevolução durante os experimentos, a NEAT desenvolveu uma estratégia mais avançada de mobilidade nas peças. Também foi demonstrada a necessidade do desenvolvimento de jogadas defensivas de a estratégias de longo prazo na coevolução [ANDERSEN; STANLEY; MIIKKULAINEN, 2002].

NEAT se destacou por estabelecer seu potencial de estratégias na entrada de novas peças no tabuleiro e ilustrou a necessidade de uma estratégia com mobilidade para derrotar um jogador que possui posicionamento forte no tabuleiro de othello [ANDERSEN; STANLEY; MIIKKULAINEN, 2002].

3.7.2 Marl/O

Marl/O é um algoritmo neuro-evolutivo aplicado ao *Super Mario World - 1990* com o objetivo de que a máquina superasse o primeiro nível do jogo. Como variável *fitness* foi utilizado o quanto longe o personagem se deslocou no cenário. O código fonte pode ser encontrado na referência [BLING, 2015].

O algoritmo neuro-evolutivo utilizado é do tipo NEAT. Esse algoritmo não evolui apenas os pesos na rede, mas toda a topologia da rede neural como um todo, sendo o método mais bem sucedido no controle de tarefas simples, pois ele consegue atingir uma rede mais eficiente com uma velocidade maior [STANLEY; MIIKKULAINEN, 2002].

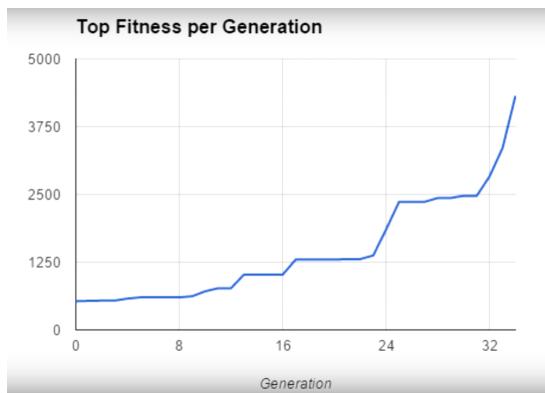


Figura 8: Gráfico da relação *fitness* e a evolução entre as gerações.

Fonte: Bling [2015]

O algoritmo obteve sucesso na geração 34 com um tempo de treinamento aproximado de 24 horas.

3.7.3 Starcraft: Brood War

Os jogos de estratégia em tempo real tornaram-se atraentes no domínio da pesquisa da IA nos últimos anos, devido ao seu dinâmico, multi-agente e ambientes multi-objetivos. Microgestão, um componente central em muitos jogos de estratégia em tempo real, envolve o controle de múltiplos agentes para realizar metas que exigem avaliação e reação rápidas e em tempo real. Nesse papel, é apresentada a aplicação e avaliação de uma técnica de Neuro-evolução na evolução dos agentes de microgestão para o jogo de estratégia em tempo real *Starcraft: Brood War* [ZHEN; WATSON, 2013].

As avaliações confirmaram a viabilidade dos algoritmos NEAT e *Real-Time Neuroevolution of augmenting topologies* (rtNEAT) em agentes em evolução para vários cenários de microgestão de *Starcraft: Brood War*. Quando os algoritmos podem funcionar sem parar, a taxa de ganhos dos agentes contra a inteligência artificial padrão do *Starcraft: Brood War* flutua altamente ao longo das gerações. No entanto, quando a evolução é in-

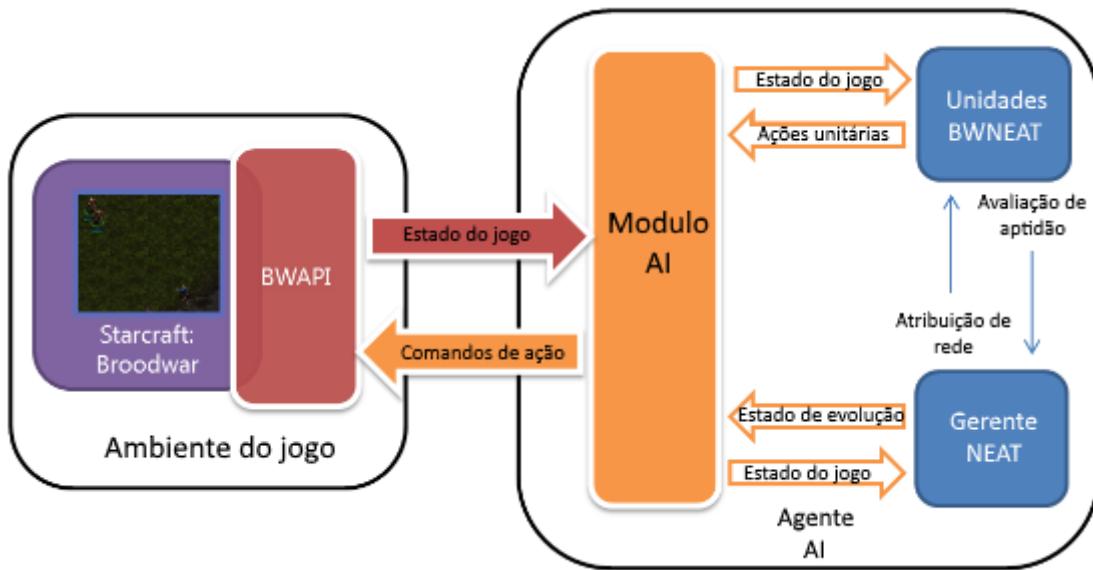


Figura 9: Uma visão geral da arquitetura do agente.

Fonte: Zhen e Watson [2013]

terrompida ao atingir um nível aceitável de desempenho, ambos os algoritmos são capazes de gerar consistentemente agentes vencedores, com a maioria de menos de 100 gerações. Cada algoritmo difere na variabilidade do desempenho em diferentes combinações de unidades. Os fatores que contribuem para a diferença de desempenho incluem a complexidade da topologia inicial da rede e a variação nos tipos de unidades. Há espaço para explorar ainda mais a complexidade da rede e a necessidade de estabelecer métodos de avaliação padronizados para avaliações de agentes de microgestão. É necessário mais estudos para adaptar essas técnicas para a implantação comercial do jogo de estratégia em tempo real, mas os resultados mostram um desempenho promissor em uma inteligência artificial neuro-evolutiva capaz de derrotar a inteligência artificial baseada em *scripts* em curto tempo de treinamento [ZHEN; WATSON, 2013].

3.7.4 Uma abordagem geral para *game playing* no video game Atari

Esse artigo aborda o desafio de aprender a jogar muitos diferentes jogos de vídeo com pouco conhecimento específico do domínio. Especificamente, ele introduz uma abordagem de neuro-evolução para o jogo geral do Atari 2600. Quatro algoritmos de neuro-evolução foram emparelhados com três representações de diferentes estados e avaliados em um conjunto de 61 jogos Atari. Os agentes de neuro-evolução representam diferentes pontos ao longo do espectro da sofisticação algorítmica - incluindo a evolução do peso em redes neurais topologicamente fixas (Neuro-evolução convencional), *Covariance matrix adaptation evolution strategy* (CMA-ES), *neuroevolution of augmenting topologies* (NEAT) e codificação de rede indireta (HyperNEAT). As representações do estado incluem uma re-

presentação de objeto da tela do jogo, os *pixels* brutos da tela do jogo e o ruído semeado (uma linha de base comparativa) [HAUSKNECHT et al., 2014].

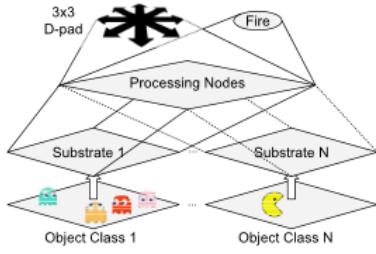


Figura 10: Arquitetura baseada em objeto.

Fonte: Hausknecht et al. [2014]

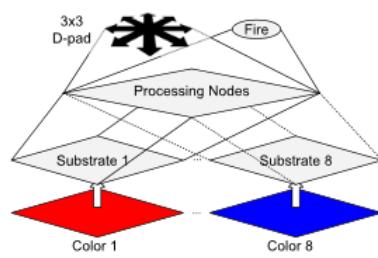


Figura 11: Arquitetura para pixels brutos.

Fonte: Hausknecht et al. [2014]

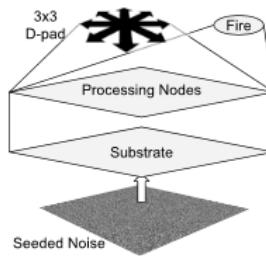


Figura 12: Arquitetura baseada em ruído semeado.

Fonte: Hausknecht et al. [2014]

Os resultados indicam que os métodos de codificação direta funcionam melhor em representações de estados compactas, ao passo que os métodos de codificação indireta (isto é, *HyperNEAT*) permitem escalar para representações de dimensões superiores (ou seja, a tela do jogo bruto). As abordagens anteriores baseadas no aprendizado da diferença temporal tiveram problemas para lidar com os grandes espaços estaduais e os gradientes de recompensas escassos, frequentemente encontrados nos jogos Atari. Neuro-evolução melhora esses problemas e as políticas evoluídas obtêm resultados de última geração, mesmo superando as altas pontuações humanas em três jogos. Esses resultados sugerem que a neuro-evolução é uma abordagem promissora para o jogo geral de videogames [HAUSKNECHT et al., 2014].

3.8 Considerações finais

Este capítulo teve como objetivo abordar o tema principal desta monografia com um maior grau de aprofundamento e detalhes desta técnica de inteligência artificial.

Como resultado do conhecimento adquirido por este estudo, está determinado qual método de algoritmo neuro-evolutivo será utilizado na implementação. O método

escolhido é o *neuroevolution of augmenting topologies* (NEAT), que leva em consideração a evolução da rede neural como um todo desde neurônios, conexões e peso da rede.

A definição de um passo a passo do algoritmo neuro-evolutivo também é importante para facilitar a implementação da solução.

Um ponto relevante apresentado é a importância de definir um ambiente ou *environment* visto na Figura 3 que é o problema que se está tentando solucionar com o algoritmo. Outro ponto de destaque é a importância da definição de métricas para observação dos resultados obtidos a cada iteração do processo evolutivo e formas de interação com o ambiente para que as saídas da rede neural tenham um resultado efetivo no ambiente.

No capítulo 4 o objetivo será o esclarecimento sobre o ambiente escolhido para aplicação do algoritmo neuro-evolutivo e porque dessa escolha.

4 Framework *EvoMan*

4.1 Considerações iniciais

Este capítulo tem o objetivo de explicar o *Environment* (ambiente) da Figura 6 e também como o *framework* *EvoMan* se encaixa nesse contexto para a implementação do algoritmo neuro-evolutivo.

Os tópicos relativos ao *framework* *EvoMan* são:

- O que é;
- Por que da utilização do *framework*;
- Como é organizada sua estrutura;
- Os possíveis tipos de simulação;
- A sua utilização e funcionamento como um todo.

O *EvoMan framework* é uma plataforma para testar algoritmos de otimização, desenvolvida na Universidade Federal do ABC em Santo André / São Paulo / Brasil, durante a pesquisa de uma dissertação de mestrado [MIRAS, 2016].

4.2 Por que o *framework* *EvoMan*

O item principal dessa monografia é a aplicação de um algoritmo neuro-evolutivo à mecânica de um jogo 2D, em que o *framework* encaixa com esse propósito.

O principal ponto para a escolha e utilização do *framework* na implementação da solução é o fato dele fornecer um ambiente estável, confiável e configurável para suprir as necessidades dos diferentes algoritmos que poderão ser testados no mesmo, passando assim uma maior confiabilidade nos testes e avaliações de um algoritmo que faz uso dessa ferramenta.

Um segundo fator é o tempo de implementação que levaria para criar do zero toda uma gama de sensores, método de controle e um modelo matemático para avaliar corretamente a população empregada ao ambiente.

4.3 Estrutura

O EvoMan foi desenvolvido em Python usando a biblioteca *pygame* e fornece 8 jogos de presa-predador inspirados no jogo eletrônico *MegaMan II* e espera simular um ambiente dinâmico, ou seja, que não apresenta sempre o mesmo comportamento [MIRAS, 2016].

4.4 Controles

Tanto a presa (jogador) quanto o predador (inimigo) podem ser controlados por um algoritmo de Inteligência Artificial (agente) ou assumir seu comportamento padrão [MIRAS, 2016].

Para o jogador, o comportamento padrão está aguardando comandos de algum dispositivo de entrada. Para o inimigo, o comportamento padrão é um ataque estático composto por regras fixas, diferente para cada inimigo [MIRAS, 2016].

4.5 Cenário e inimigos

Cada um dos 8 inimigos possui um tipo de desafio diferente, seja com diferentes tipos de ataques, comportamento do inimigo ou variações de como o cenário foi criado.

As variações de cenários disponíveis podem se distinguir pelas seguintes características:

- Dois cenários com elevações no cenário;
- Um cenário aquático fazendo com que os movimentos dos personagens sejam diferentes do comum e também possui obstáculos na parte superior que causam dano no personagem;
- Cinco cenários simples e lineares.

As variações de inimigos dependem do cenário em que se está simulando. As características dos inimigos variam nos seguintes aspectos:

- Inimigos que disparam um projétil simples;
- Inimigos que disparam múltiplos projéteis de comportamento variável;
- Inimigos que se movimentam de forma defensiva;
- Inimigos que se movimentam de forma mais agressiva em direção ao personagem;



Figura 13: Inimigos disponíveis para treino dos agentes autônomos.

Fonte: Miras [2016]

Cada um dos 8 inimigos disponíveis está atrelado a um dos 8 cenários disponíveis. Pode-se enxergar cada um como um problema a ser resolvido dentro do fluxo do algoritmo. É justamente essa variação combinada de cenário e inimigo que transforma o *framework* EvoMan em uma escolha ótima para a implementação do algoritmo.

4.6 Tipos de simulações

A combinação desses métodos de controle podem gerar uma série de combinações de simulação para testar situações de diferentes formas de aprendizado para vários algoritmos que podem vir a serem escritos [MIRAS, 2016].

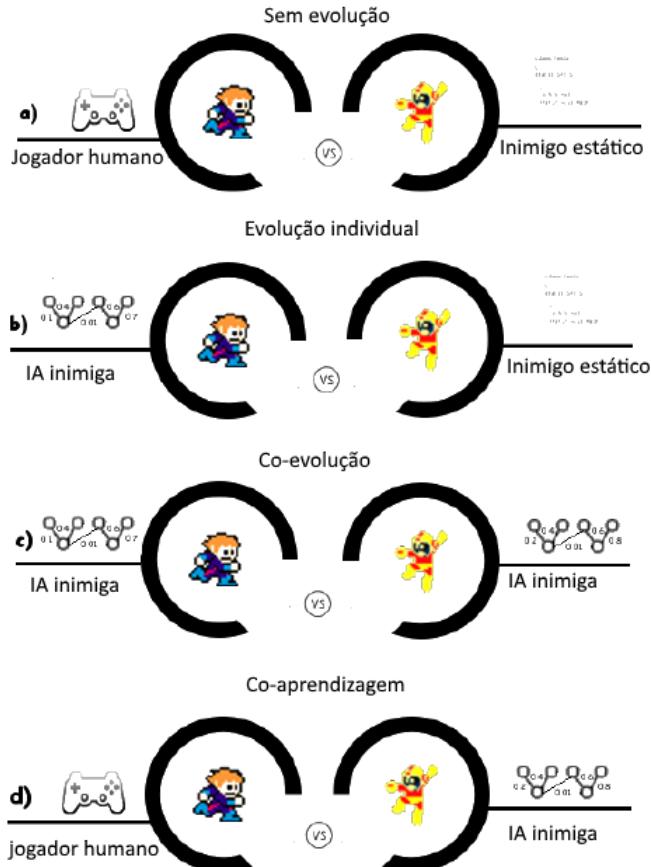


Figura 14: Possíveis combinações de controle para simulação.

Fonte: Miras [2016]

Segundo a documentação do ambiente escrita por Miras [2016] pode-se iniciar a simulação das seguintes formas:

- Modo em que um jogador humano irá controlar a presa e um algoritmo estático interno na simulação irá controlar o caçador;
- Modo em que um agente autônomo externo irá controlar a presa contra o caçador;
- Modo em que tanto a presa quanto o caçador irão ser controlados por agentes autônomos externos à simulação;
- Modo em que um jogador humano irá controlar a presa e um agente autônomo irá controlar o caçador.

4.7 Objetivos de treinamento

É possível configurar o ambiente EvoMan para o treinamento de dois tipos de agentes autônomos com objetivos diferentes: agente especialista ou generalista [Miras, 2016].

O agente do tipo especialista será aquele que realizará o treinamento apenas contra um dos tipos dos 8 inimigos disponíveis, se especializando então somente em desafiar um tipo de inimigo [MIRAS, 2016].

O agente do tipo generalista será aquele que realizará o treinamento desafiando todos os 8 tipos de inimigos disponíveis, tornando-se capaz de desafiar todos os tipos de inimigos disponíveis no *framework* [MIRAS, 2016].



Figura 15: Possíveis configuração de objetivos.

Fonte: Miras [2016]

4.8 Sensores

Segundo a documentação escrita por Miras [2016], o *framework* EvoMan disponibiliza acesso a uma série de 20 sensores sendo eles:

- 16 sensores com as diferentes distâncias de projéteis;
- 2 sensores de distância com o inimigo;
- 2 sensores da direção dos *sprites*.

Estes 20 sensores são utilizados como entrada na rede neural que será treinada pelo algoritmo neuro-evolutivo.

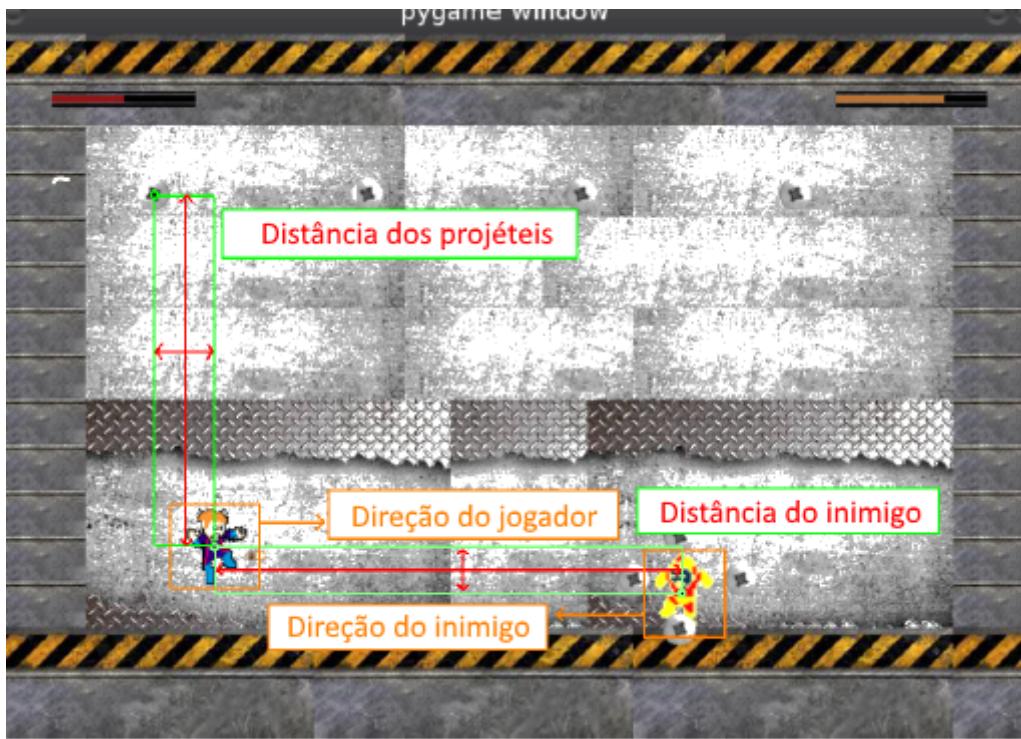


Figura 16: Sensores.

Fonte: Miras [2016]

4.9 Fitness

Como saída ao final de uma execução de simulação, o *framework* fornece o valor qualitativo de como o agente autônomo se saiu, ou seja, retorna um valor para ser utilizado como o atributo *fitness*, possibilitando assim qualificarmos cada um dos genomas que são testados.

Como citado anteriormente, podemos gerar dois tipos de simulações para finalidades diferentes, uma para treinar um agente especialista e outro para treinar um agente generalista.

Existem duas fórmulas matemáticas para o cálculo do *fitness*, cada uma aplicada a um dos dois tipos de simulação, sendo uma fórmula para avaliar o agente especialista e outra fórmula para avaliar o agente generalista. Ambos os métodos podem ser substituídos caso seja desejado por quem estiver implementando uma solução [MIRAS, 2016].

A fórmula para agentes especialistas:

$$\text{fitness} = \gamma * (100 - e_e) + \alpha * e_p - \log t , \quad (4.1)$$

sendo e_e e e_p os medidores da vida do inimigo e do jogador respectivamente, variando o valor de 0 a 100; t é o número de etapas de tempo executadas em uma

simulação; γ e α são constantes que assumem os valores de 0.9 e 0.1 respectivamente [MIRAS, 2016].

A fórmula para agentes generalistas:

$$\text{fitness}' = \bar{f} - \sigma , \quad (4.2)$$

sendo $\text{fitness}'$ o fitness consolidado pelo cálculo de \bar{f} e σ , que são a média e o desvio padrão contra os n inimigos escolhidos para realizar o processo de treinamento [MIRAS, 2016].

4.10 Considerações finais

Este capítulo mostrou o funcionamento do *Environment* (ambiente) da Figura 6 e também como o *framework* EvoMan se encaixa nesse contexto para a implementação do algoritmo neuro-evolutivo.

Os esclarecimentos sobre a finalidade do *framework* ajudaram a ponderar a escolha de utilização do mesmo na implementação.

Os pontos levantados sobre o tema do capítulo ajudaram a entender melhor alguns pontos que serão utilizados na implementação:

- Estrutura - qual linguagem de programação e qual a mecânica de jogo 2D em que o *framework* foi inspirado;
- Controles - como funciona a mecânica de presa e predador nos ambientes disponíveis;
- Inimigos - entender melhor quais são os problemas, ou seja, os inimigos em que o agente autônomo irá aprender a ganhar;
- Simulações - quais os tipos de simulações são possíveis configurar e as variações possíveis;
- Objetivos - as diferenças entre especialista e generalista no processo de treinamento dentro do *framework*;
- Sensores - quais os sensores disponíveis para serem usados de entrada na rede neural que passará pelo processo de treinamento ou teste;
- *Fitness* - como funciona o critério de avaliação do *framework* para agentes especialistas e generalistas.

No capítulo 5 é apresentado o algoritmo implementado, a linguagem utilizada, as funcionalidades da biblioteca utilizada para criação e manipulação da população, redes neurais e todo processo evolutivo além de alguns resultados.

5 Implementação do algoritmo

5.1 Considerações iniciais

O objetivo deste capítulo é a apresentação da solução desenvolvida, testes realizados, tecnologias envolvidas e resultados. Os tópicos abordados são os seguintes:

- A lógica do algoritmo implementado;
- A linguagem de programação utilizada e suas características;
- A biblioteca chave utilizada para a implementação do algoritmo;
- Explicação sobre trechos de código mais relevantes para a implementação;
- Testes e resultados.

5.1.1 Linguagem

Python foi a linguagem escolhida para a implementação da solução devido ao seu desempenho e variedade de aplicações no campo da inteligência artificial encontradas no decorrer das pesquisas e desenvolvimento da revisão bibliográfica.

Segundo Venners [2003], algumas das características do Python são:

- Linguagem de programação com nível de abstração considerado pelos tipos de classificação como alto nível;
- Interpretada;
- Pensada para implementação de *scripts*;
- Orientada a objetos;
- Funcional;
- Possui tipagem dinâmica e forte.

Foi lançada por Guido van Rossum no ano de 1991, hoje possui uma fundação chamada *Python software foundation* que possui um modelo de desenvolvimento aberto com a comunidade, ficando encarregada somente pelo gerenciamento da evolução da linguagem [VENNERS, 2003].

Segundo Python Software Foundation [2003], a missão da Python software foundation é promover, proteger e avançar a linguagem de programação Python, e para dar suporte e facilitar o crescimento da diversidade da comunidade internacional da linguagem de programação Python.

Devido as suas características, no decorrer de sua evolução, além de uso para inteligência artificial, foi utilizada fortemente para a implementação de processadores de textos, dados científicos e criação de CGI's (*Common Gateway Interface*) para páginas WEB [VENNERS, 2003].

5.2 Algoritmo

A técnica de neuro-evolução escolhida para o desenvolvimento da solução foi o NEAT (*Neuroevolution of augmenting topologies*) que visa a evolução da estrutura da rede neural como um todo [STANLEY; MIIKKULAINEN, 2002]. Esse método foi apresentado no artigo *Evolving Neural Networks through Augmenting Topologies* por Stanley e Miikkulainen [2002].

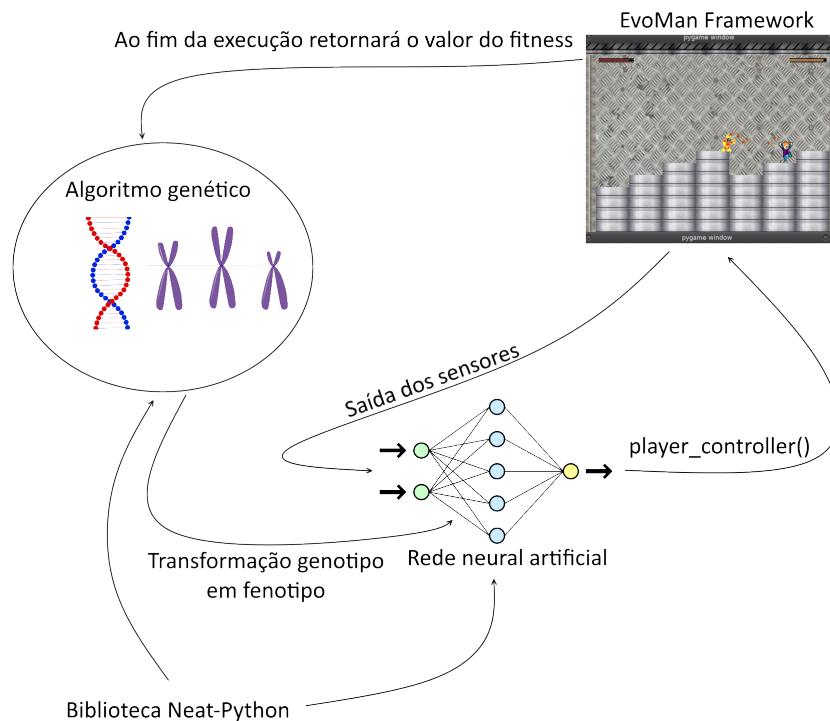


Figura 17: Representação visual do algoritmo implementado.

Fonte: O autor

A Figura 17 é a representação visual da solução implementada.

5.2.1 Biblioteca *NEAT-Python*

NEAT-Python é uma implementação Python pura do NEAT, sem dependências diferentes da biblioteca padrão do Python [STANLEY, 2015].

Segundo Stanley [2015], na implementação atual de NEAT-Python, uma população de genomas individuais é mantida. Cada genoma contém dois conjuntos de genes que descrevem como construir uma rede neural artificial:

- Nós genes, cada um dos quais especifica um único neurônio;
- Os genes de conexão, cada um especificando uma única conexão entre os neurônios.

Para evoluir uma solução para um problema, o usuário deve fornecer uma função de *fitness* que calcula um único número real que indique a qualidade de um genoma individual: uma melhor capacidade de resolver o problema significa uma pontuação maior. O algoritmo progride através de um número de gerações especificado pelo usuário, sendo cada geração produzida por reprodução (seja sexual ou assexuada) e mutação dos indivíduos mais aptos da geração anterior [STANLEY, 2015].

As operações de reprodução e mutação podem adicionar nós e / ou conexões a genomas, de modo que o algoritmo procede de genomas (e as redes neurais que produzem) podem tornar-se cada vez mais complexas. Quando o número predefinido de gerações é alcançado, ou quando pelo menos um indivíduo (para uma função de critério de aptidão máxima, outros são configuráveis) excede o limite de aptidão especificado pelo usuário, o algoritmo termina [STANLEY, 2015].

Uma dificuldade nessa configuração é a implementação do *crossover* - como se faz um cruzamento entre duas redes de diferentes estruturas? O NEAT lida essa situação acompanhando as origens dos nós com um número de identificação (novos e maiores números são gerados para cada nó adicional). Aqueles derivados de um antepassado comum (que são homólogos) são combinados para *crossover*, e as conexões são compatíveis se os nós que eles conectam possuam ancestralidade comum. Existem variações de exatamente como isso é feito dependendo da implementação do NEAT. A implementação utilizada é como descrita anteriormente[STANLEY, 2015].

Uma outra dificuldade potencial é que uma mutação estrutural - em oposição a mutações, por exemplo, o peso das conexões - como a adição de um nó ou conexão pode, ao ser promissor para o futuro, ser perturbador no curto prazo (até que tenha sido ajustado por mutações menos perturbadoras). NEAT trata essa situação dividindo os genomas em espécies, que têm uma distância genômica próxima devido à similaridade, tendo então a competição mais intensa dentro das espécies, e não entre as espécies (compartilhamento físico). A distância genômica é medida usando uma combinação do número de nós e

conexões não homólogas com medidas de quanto os nós e conexões homólogas divergiram desde sua origem comum (Os nós e conexões não homólogos são denominados disjuntos ou excessivos, dependendo se os números são do mesmo alcance ou além desse intervalo, como a maioria das implementações NEAT, este não faz distinção entre os dois.) [STANLEY, 2015].

5.2.2 Instância de ambiente com *framework* EvoMan

O primeiro passo para criar uma instância de simulação é utilizando a classe *Environment* e passando as devidas configurações pelo parâmetro da classe. O exemplo de trecho de código a seguir mostra os atributos de configurações utilizados nos testes, dependendo do teste os valores dos atributos foram modificados.

```
env = Environment(multiplemode = 'no',
                   enemies = [1], playermode = 'ai',
                   player_controller = player_controller(),
                   speed = 'fastest',
                   randomini = 'yes')
```

Cada um dos atributos passados por parâmetro possui uma função:

- O atributo *multiplemode* pode ser configurado com uma *string* de valor ‘no’ ou ‘yes’, indicando qual será o tipo objetivo para a simulação, caso seja configurado para ‘no’ a execução será com o objetivo do treinamento de um agente especialista em algum dos 8 inimigos disponíveis e será avaliado segundo a função 4.1, caso seja configurado para ‘yes’ a execução será com o objetivo do treinamento de um agente generalista e será avaliado segundo a função 4.2;
- O *enemies* é um atributo usado para indicar qual ou quais inimigos carregar através de um *array* variando os valores de cada posição no *array* de 1 a 8. Para as simulações de especialistas se passam apenas um *array* com uma posição e para os generalistas um *array* com 2 ou mais posições dependendo dos inimigos e ordem desejada;
- O *player_controller* é uma classe que deve ser substituída pelo método que usará a rede neural para controlar a presa na simulação e será avaliado durante a simulação. Este método será melhor explicado na seção de estrutura de controle;
- O *speed* é um atributo utilizado para configurar a velocidade em que a simulação é executada, pode ser configurada com dois valores no formato de *string*: o primeiro ‘normal’ executando a simulação em uma velocidade de 30 quadros por segundo e ‘fastest’ executando a simulação sem limitação de quadros por segundo acelerando o processo de treinamento;

- O atributo *randomini* pode ser configurado com dois valores em formato de *string*, sendo ‘yes’ ou ‘no’, caso seja configurada com o valor de ‘yes’ o inimigo irá nascer em partes aleatórias do mapa aumentando o nível de dificuldade para o processo de treinamento, caso seja configurado com o valor ‘no’ o inimigo irá nascer sempre na mesma posição ao inicio de cada simulação.

5.2.3 Estrutura de controle

```
# Implementacao da estrutura de controle dos sprites do jogador
class player_controller(Controller):

    # params - a lista de sensores com
    # os valores atuais dos mesmos
    # cont - o objeto do tipo net ou
    # seja o objeto “rede neural”
    def control(self, params, cont):

        # recebe o objeto do tipo net para ser avaliado
        net = cont

        # retorna o resultado de saida da rede neural
        # em forma de uma lista para a variavel output
        output = net.activate(params)

        # toma as decisoes de acao para o sprite do jogador
        if output[0] > 0.5:
            left = 1
        else:
            left = 0

        if output[1] > 0.5:
            right = 1
        else:
            right = 0

        if output[2] > 0.5:
            jump = 1
        else:
            jump = 0
```

```

if output [3] > 0.5:
    shoot = 1
else:
    shoot = 0

if output [4] > 0.5:
    release = 1
else:
    release = 0

# dado como retorno uma lista de booleanos indicando
# as ações para o sprite do jogador
return [left , right , jump , shoot , release]

```

O foco dessa classe é a utilização do fenótipo (rede neural) passado pelo parâmetro *cont* para a execução das ações do personagem dentro da simulação, podendo assim avaliar o desempenho da rede neural.

Os comentários distribuídos pelo trecho de código da função dão um melhor entendimento sobre o funcionamento e a lógica de execução dessa funcionalidade.

5.2.4 Avaliação de genomas

A avaliação de genomas é feito pelo método (*eval_genomes*). O foco deste método é realizar o processo de transformação de um genoma em fenótipo, ou seja, rede neural e logo em seguida executar o método *simula* que fará a execução de uma simulação utilizando este fenótipo para o controle do personagem e retornando o valor do *fitness* ao fim da execução como resultado da simulação.

```

def eval_genomes (genomes , config):
    # for dedicado a percorrer toda a população de genomas
    for genome_id , genome in genomes:

        # transforma o genoma que está sendo avaliado
        # em fenótipo ou seja uma rede neural
        net = neat . nn . RecurrentNetwork . create (genome , config)

        # Método simula é executado e retorna o valor de
        # fitness da rede neural
        #
        # genome.fitness é um atributo do objeto genoma

```

```

# que tem a finalidade de guardar
# o resultado do fitness deste genoma
genome.fitness = simula(env, net)

```

O método *simula* é responsável por pegar a rede neural e a instância de simulação configurada antecipadamente e executar o método *play()* do ambiente que irá colocar a rede neural para ser avaliada, como retorno se dá o *fitness* resultante da avaliação.

```

def simula(env, x):
    # f = fitness result
    # p = player life result
    # e = enemy life result
    # t = time result
    f, p, e, t = env.play(pcont=x)
    return f

```

5.2.5 Configuração da população de redes neurais

Segundo Stanley [2015], o arquivo de configuração está no formato descrito na documentação do *configparser* do Python como “uma linguagem de analisador de arquivo de configuração básica que fornece uma estrutura semelhante à que você encontraria nos arquivos INI do *Microsoft Windows*”.

O arquivo de configuração está em várias seções, das quais pelo menos um é necessário. No entanto, não há requisitos para ordenar dentro dessas seções, ou para ordenar as seções em si [STANLEY, 2015].

```

[NEAT]
fitness_criterion = max
fitness_threshold = 95
pop_size = 25
reset_on_extinction = False

[DefaultStagnation]
species_fitness_func = max
max_stagnation = 20
species_elitism = 3

[DefaultReproduction]
elitism = 0
survival_threshold = 0.2
min_species_size = 2

```

```
[ DefaultSpeciesSet ]
compatibility_threshold      = 3.0

[ DefaultGenome ]
# opções de ativação do nó
activation_default      = sigmoid
activation_mutate_rate   = 0.0
activation_options       = sigmoid

# opções de agregação do nó
aggregation_default     = sum
aggregation_mutate_rate = 0.0
aggregation_options     = sum

# opções de polarização (bias) do nó
bias_init_mean           = 0.0
bias_init_stdev          = 1.0
bias_init_type            = gaussian
bias_max_value           = 30.0
bias_min_value           = -30.0
bias_mutate_power        = 0.5
bias_mutate_rate          = 0.7
bias_replace_rate         = 0.1

# opções de compatibilidade do genoma
compatibility_disjoint_coefficient = 1.0
compatibility_weight_coefficient   = 0.5

# taxas de adição e remoção de conexão
conn_add_prob             = 0.5
conn_delete_prob           = 0.5

# opções de habilitação de conexão
enabled_default           = True
enabled_mutate_rate        = 0.01

feed_forward               = False
initial_connection          = full
```

```

# taxas de adição e remoção de nós
node_add_prob          = 0.2
node_delete_prob        = 0.2

# parâmetros de rede
num_hidden              = 0
num_inputs               = 20
num_outputs              = 5

# opções de resposta do nó
response_init_mean      = 1.0
response_init_stdev     = 0.0
response_init_type       = gaussian
response_max_value       = 30.0
response_min_value       = -30.0
response_mutate_power   = 0.0
response_mutate_rate    = 0.0
response_replace_rate   = 0.0

# opções de peso da conexão
weight_init_mean         = 0.0
weight_init_stdev        = 1.0
weight_init_type         = gaussian
weight_max_value         = 30
weight_min_value         = -30
weight_mutate_power     = 0.5
weight_mutate_rate       = 0.8
weight_replace_rate      = 0.1

```

5.2.6 Execução

Este trecho de código é o responsável por realizar a execução de todo o processo, ele vai desde o carregamento do arquivo de configuração à geração dos gráficos de resultados que serão apresentados na seção de testes e resultados.

```

def run(config_file):
    # carrega o arquivo de configuração realizando os “parsers”
    # necessários para coletar as configurações.

```

```
config = neat.Config(neat.DefaultGenome,
                     neat.DefaultReproduction,
                     neat.DefaultSpeciesSet,
                     neat.DefaultStagnation,
                     config_file)

# Cria uma populacao de genomas baseadas nas configuracoes
# setadas no arquivo
p = neat.Population(config)

# Adiciona algumas configuracoes de saidas no terminal para
# gerar pequenos relatorios durante a execucao.
p.add_reporter(neat.StdOutReporter(True))
stats = neat.StatisticsReporter()
p.add_reporter(stats)

# informa no console e salva um arquivo com o estado atual
# da populacao de genotipos atual para evitar
# perdas durante a execucao caso ocorra uma falha de
# energia por exemplo.
p.add_reporter(neat.Checkpointer(5))

# Executa N vezes o metodo de qualificacao de fitness para
# toda a populacao parametro 1 passa-se o metodo e o
# segundo a quantidade de geracoes em que sera executada.
# Caso atinja-se o fitness antes do valor de geracoes
# estimado a execucao e interrompida visto que o objetivo
# primario e sempre atingir o valor de fitness desejado
winner = p.run(eval_genomes, 100)

# Mostra qual o genoma obteve o maior fitness dentre as N
# execucoes de avaliacao e ou o melhor genoma que atingiu
# o valor igual ou superior ou desejado e determinado no
# arquivo de configuracao.
print('\nBest_genome:\n{}' .format(winner))

# Transforma o genoma e fenotipo para uma analise visual
# do resultado dele
print('\nOutput: ')
```

```

winner_net = neat.nn.RecurrentNetwork.create(winner, config)

# lista com os nomes das entradas da rede neural
# com os valores negativos
# lista com os nomes das saidas da rede neural
# com os valores positivos
node_names = {-1: 'dintancia_projetil_1',
-2: 'distancia_projetil_2', -3: 'distancia_projetil_3',
-4: 'distancia_projetil_4', -5: 'distancia_projetil_5',
-6: 'distancia_projetil_6', -7: 'distancia_projetil_7',
-8: 'distancia_projetil_8', -9: 'distancia_projetil_9',
-10: 'distancia_projetil_10', -11: 'distancia_projetil_11',
-12: 'distancia_projetil_12', -13: 'distancia_projetil_13',
-14: 'distancia_projetil_14', -15: 'distancia_projetil_15',
-16: 'distancia_projetil_16', -17: 'distancia_inimigo_1',
-18: 'distancia_inimigo_2', -19: 'direcao_sprites_1',
-20: 'direcao_sprites_2', 0: 'esquerda', 1: 'direita',
2: 'pular', 3: 'atirar', 4: 'release'}

# desenha uma representacao visual do fenotipo "campeao"
visualize.draw_net(config, winner, True,
                    node_names=node_names)

# desenha um grafico de duas dimensoes com y sendo o
# fitness e x sendo as geracoes,
# demonstrando assim a evolucao do nivel de fitness
# pelo passar das geracoes
visualize.plot_stats(stats, ylog=False, view=True)

# desenha visualmente a variacao e geracao de novas
# especies com o passar das geracoes
visualize.plot_species(stats, view=True)

if __name__ == '__main__':
    # Aqui e onde toda a "acao" comeca. e determinado o caminho
    # para o arquivo de configuracao.
    # Este arquivo de configuracao sera manipulado e se estiver
    # com as configuracoes
    # sem conflitos ira executar normalmente.

```

```

local_dir = os.path.dirname(__file__)
config_path = os.path.join(local_dir, 'config-Neat-Evoman')
run(config_path)

```

5.3 Testes e Resultados

Para realizar os testes do algoritmo foram treinados 8 agentes especialistas, sendo um agente especialista para cada um dos 8 inimigos disponíveis pelo *framework* EvoMan, variando o tamanho da população de genomas entre 10, 25, 50, 100 e 150, resultando num total de 40 testes de agentes especialistas.

Todos os testes foram realizados em um período de 100 gerações com um *fitness* de 99.5 propositalmente alto para garantir o processo evolutivo sem interrupções até a geração de numero 100.

Como resultados foram geradas 2 informações:

- Gráfico da evolução do *fitness* no decorrer das gerações. Observações:
 - A linha vermelha representa o maior *fitness* de cada geração;
 - A linha azul representa a media do *fitness* de cada população;
 - A linha verde é o desvio padrão do *fitness*.
- Gráfico da variação e sobrevivência de espécies do decorrer de cada geração;

5.3.1 Especialista inimigo 1

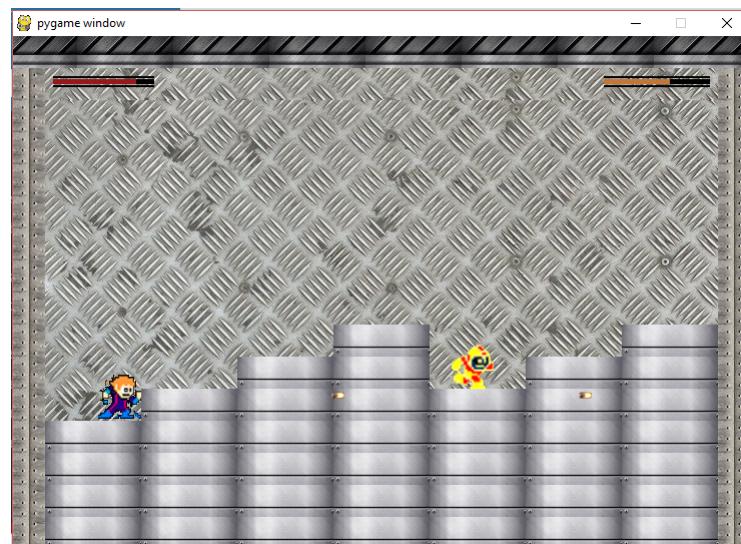


Figura 18: Inimigo e cenário utilizado para o agente especialista 1

Fonte: Miras [2016]

Os resultados do especialista 1 são exibidos nas próximas Figuras 19, 20, 21, 22 e 23. Somente o teste com população de 150 indivíduos gerou mais de uma espécie no decorrer das gerações e pode ser observado na Figura 24.

5.3.1.1 Resultados com a população de tamanho 10

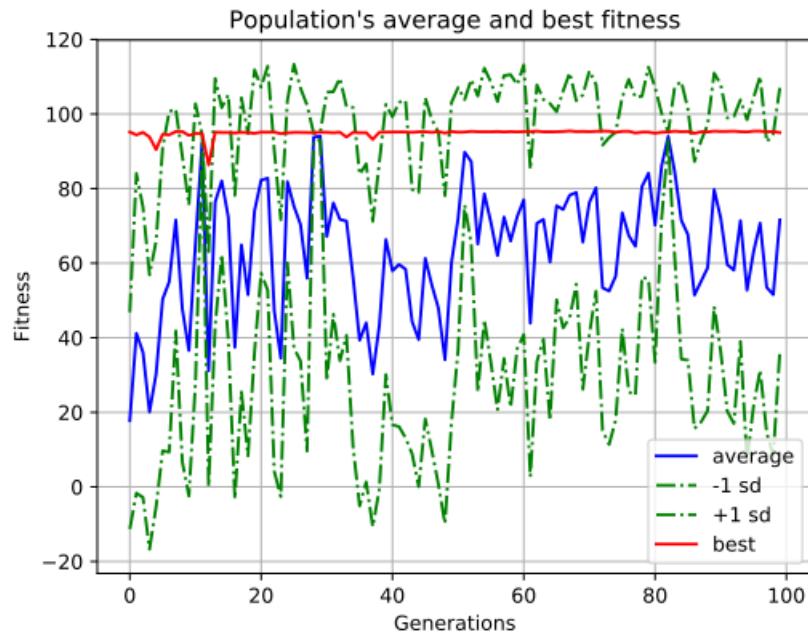


Figura 19: Evolução do *fitness* especialista inimigo 1 com população de tamanho 10.

Fonte: O autor

5.3.1.2 Resultados com a população de tamanho 25

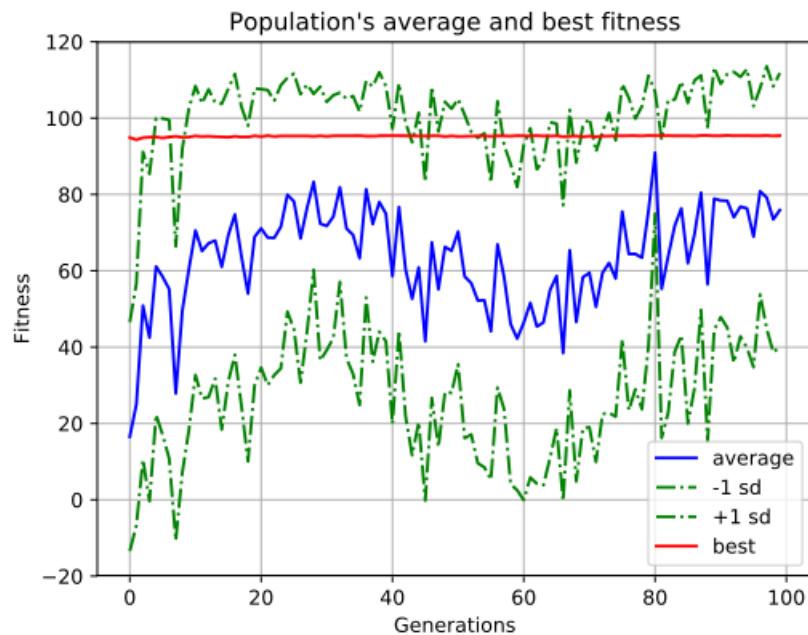


Figura 20: Evolução do *fitness* especialista inimigo 1 com população de tamanho 25.

Fonte: O autor

5.3.1.3 Resultados com a população de tamanho 50

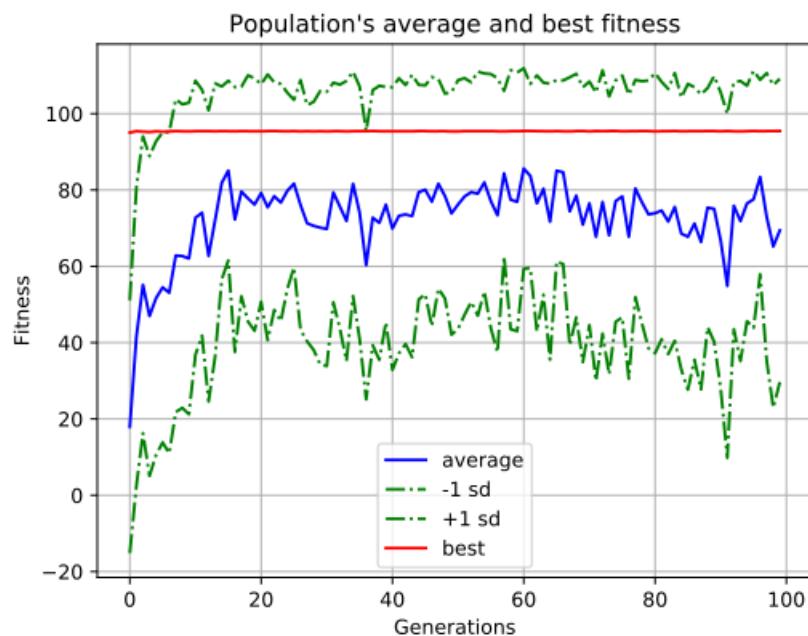


Figura 21: Evolução do *fitness* especialista inimigo 1 com população de tamanho 50.

Fonte: O autor

5.3.1.4 Resultados com a população de tamanho 100

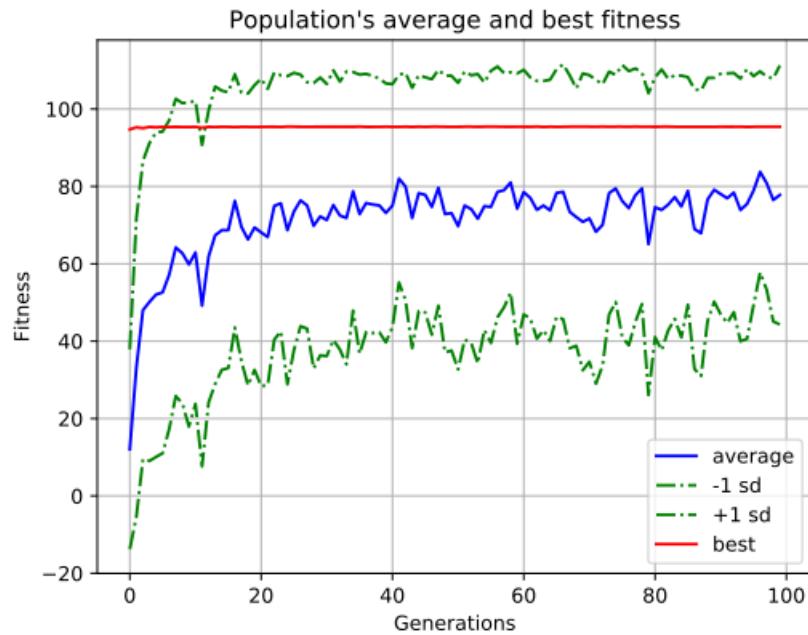


Figura 22: Evolução do *fitness* especialista inimigo 1 com população de tamanho 100.

Fonte: O autor

5.3.1.5 Resultados com a população de tamanho 150

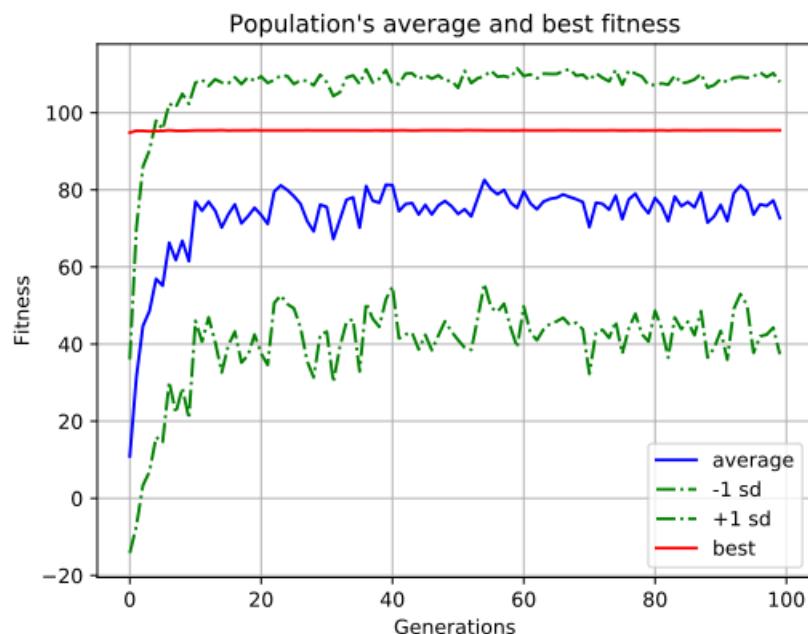


Figura 23: Evolução do *fitness* especialista inimigo 1 com população de tamanho 150.

Fonte: O autor

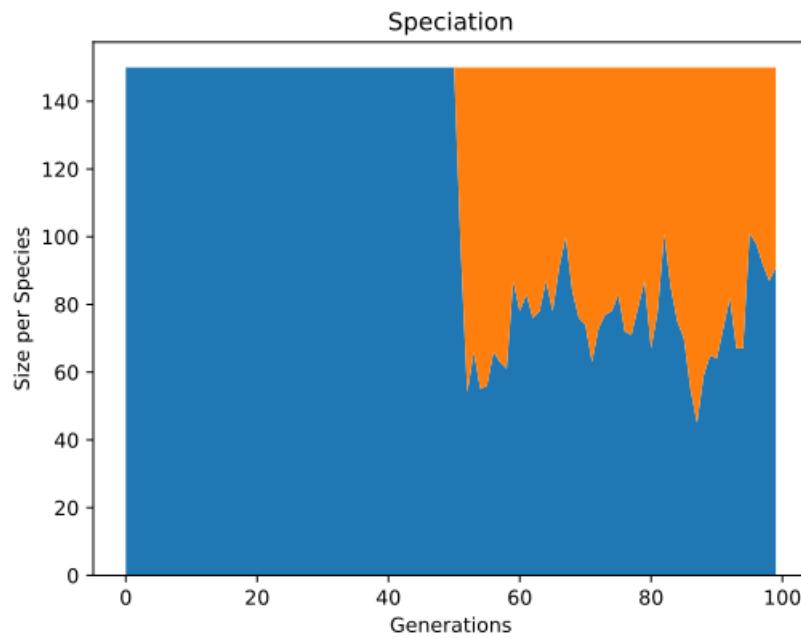


Figura 24: Variedade de espécies do *fitness* especialista inimigo 1 com população de tamanho 150.

Fonte: O autor

5.3.2 Especialista inimigo 2



Figura 25: Inimigo e cenário utilizado para o agente especialista 2

Fonte: Miras [2016]

Os resultados do especialista 2 são exibidos nas próximas Figuras 26, 27, 28, 29 e 30. Somente o teste com população de 150 indivíduos gerou mais de uma espécie no decorrer das gerações e pode ser observado na Figura 31.

5.3.2.1 Resultados com a população de tamanho 10

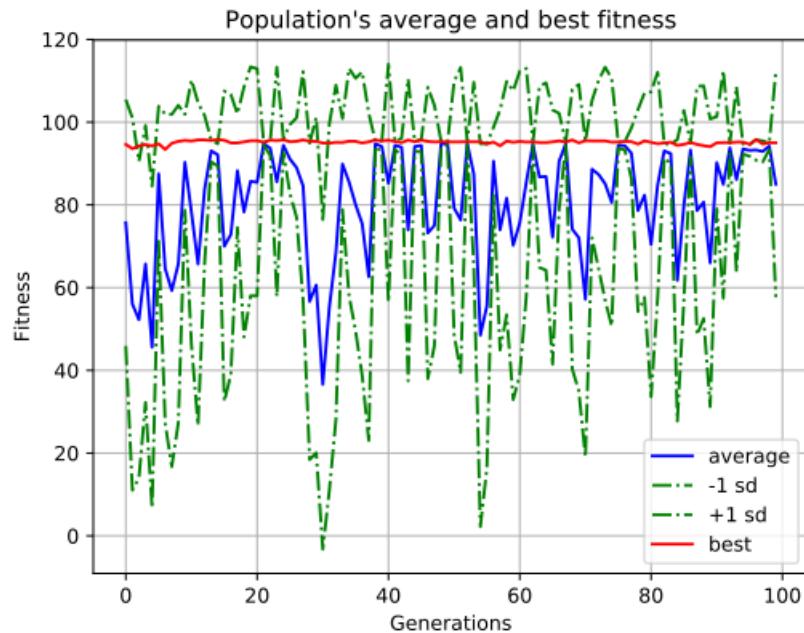


Figura 26: Evolução do *fitness* especialista inimigo 2 com população de tamanho 10.

Fonte: O autor

5.3.2.2 Resultados com a população de tamanho 25

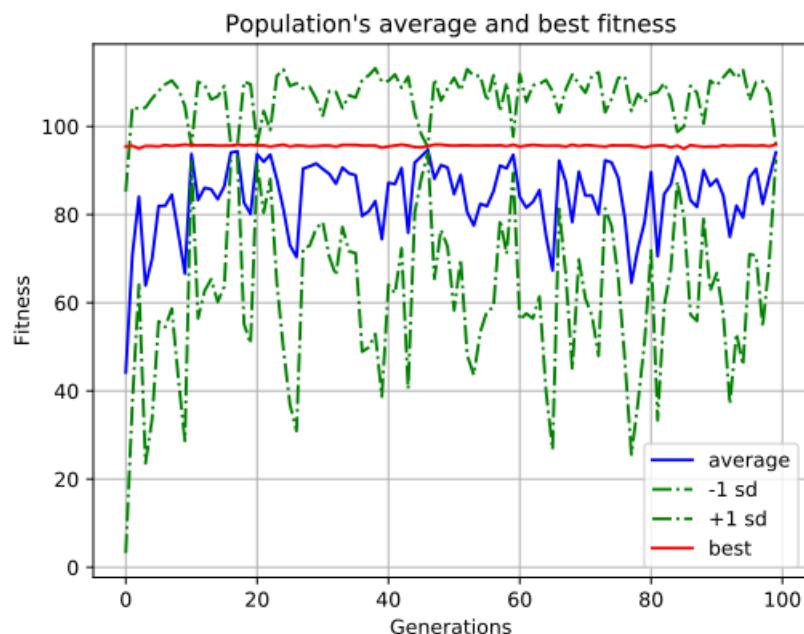


Figura 27: Evolução do *fitness* especialista inimigo 2 com população de tamanho 25.

Fonte: O autor

5.3.2.3 Resultados com a população de tamanho 50

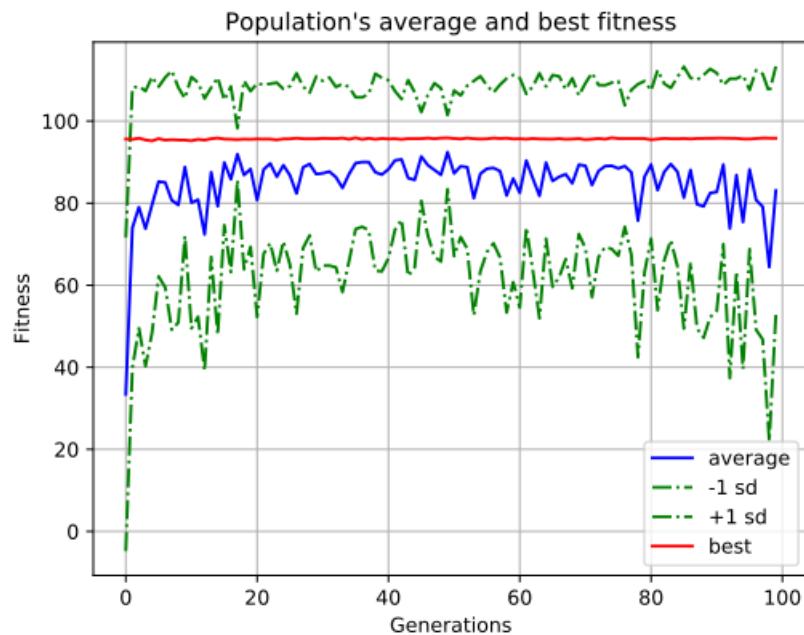


Figura 28: Evolução do *fitness* especialista inimigo 2 com população de tamanho 50.

Fonte: O autor

5.3.2.4 Resultados com a população de tamanho 100

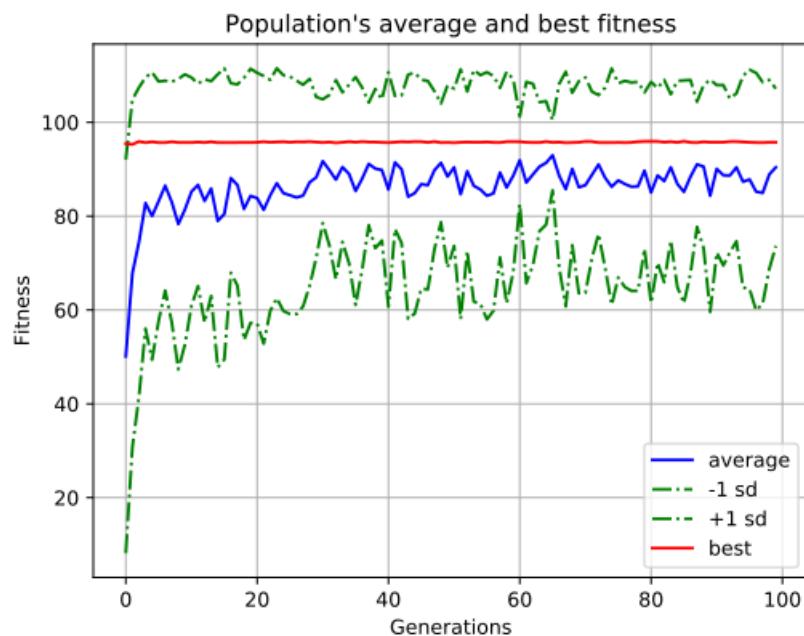


Figura 29: Evolução do *fitness* especialista inimigo 2 com população de tamanho 100.

Fonte: O autor

5.3.2.5 Resultados com a população de tamanho 150

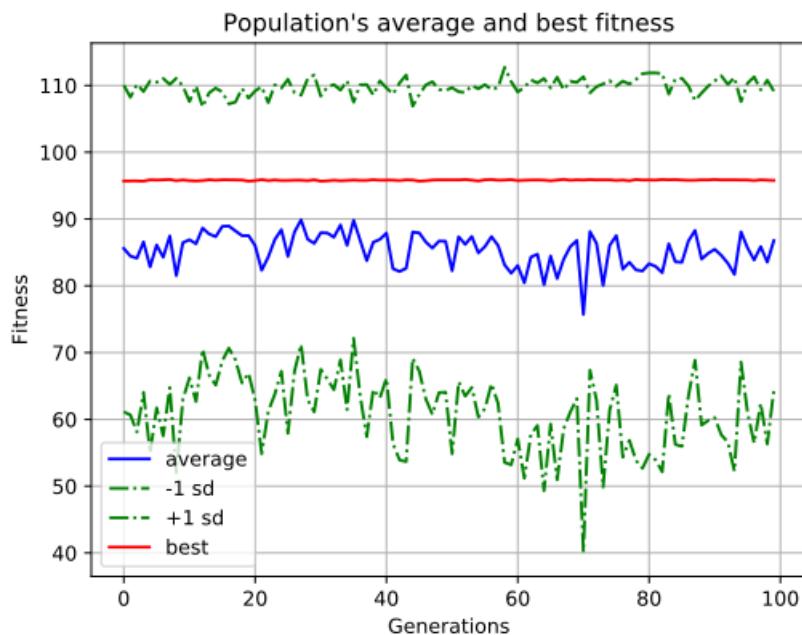


Figura 30: Evolução do *fitness* especialista inimigo 2 com população de tamanho 150.

Fonte: O autor

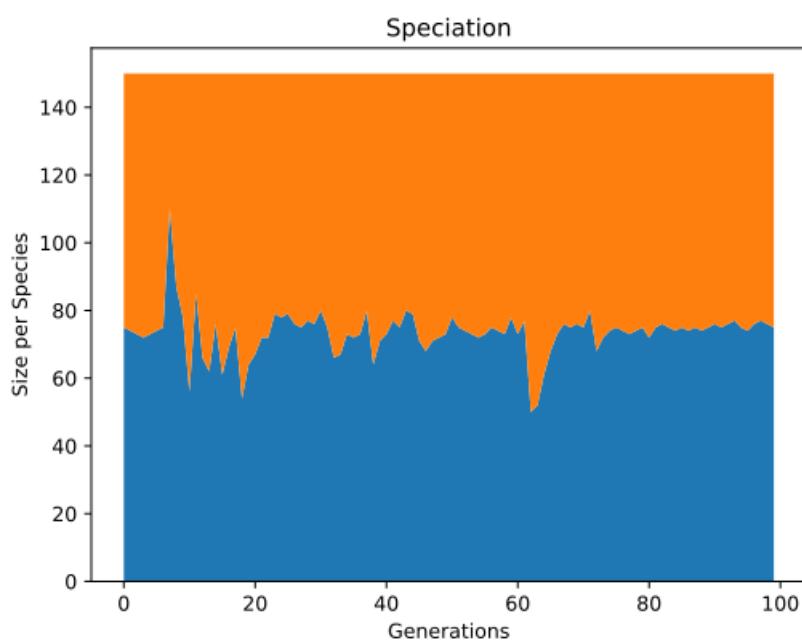


Figura 31: Variedade de espécies do *fitness* especialista inimigo 2 com população de tamanho 150.

Fonte: O autor

5.3.3 Especialista inimigo 3



Figura 32: Inimigo e cenário utilizado para o agente especialista 3

Fonte: Miras [2016]

Os resultados do especialista 3 são exibidos nas próximas Figuras 33, 34, 35, 36 e 38. Somente os testes com população de 100 e 150 indivíduos geraram mais de uma espécie no decorrer das gerações e podem ser observados nas Figuras 37 e 39.

5.3.3.1 Resultados com a população de tamanho 10

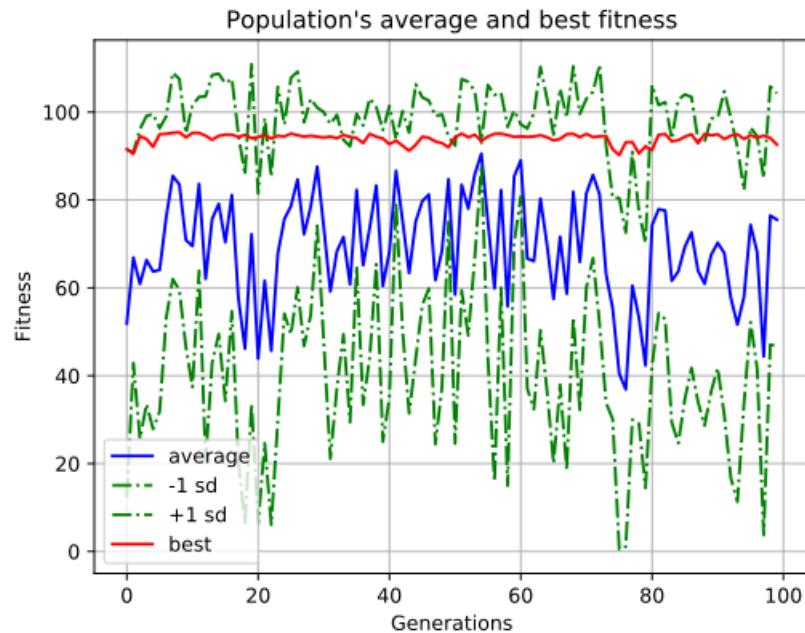


Figura 33: Evolução do *fitness* especialista inimigo 3 com população de tamanho 10.

Fonte: O autor

5.3.3.2 Resultados com a população de tamanho 25

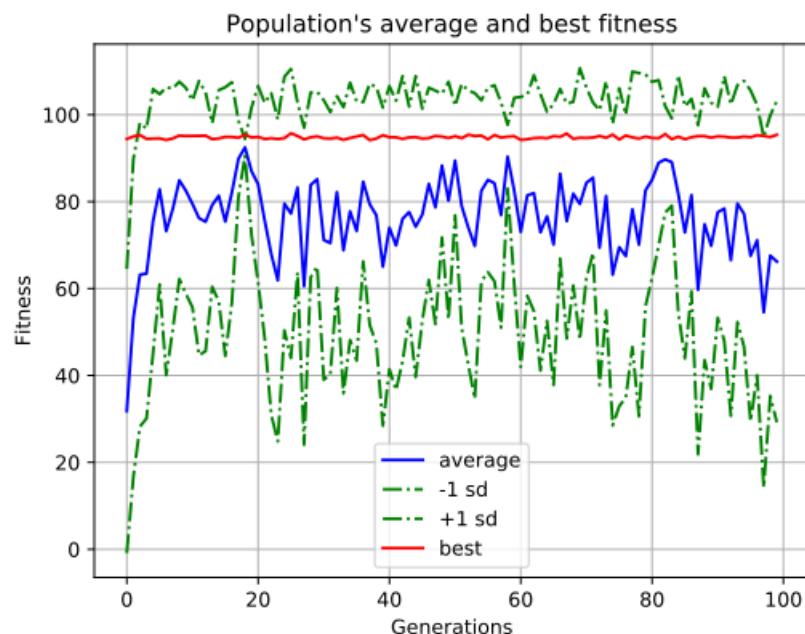


Figura 34: Evolução do *fitness* especialista inimigo 3 com população de tamanho 25.

Fonte: O autor

5.3.3.3 Resultados com a população de tamanho 50

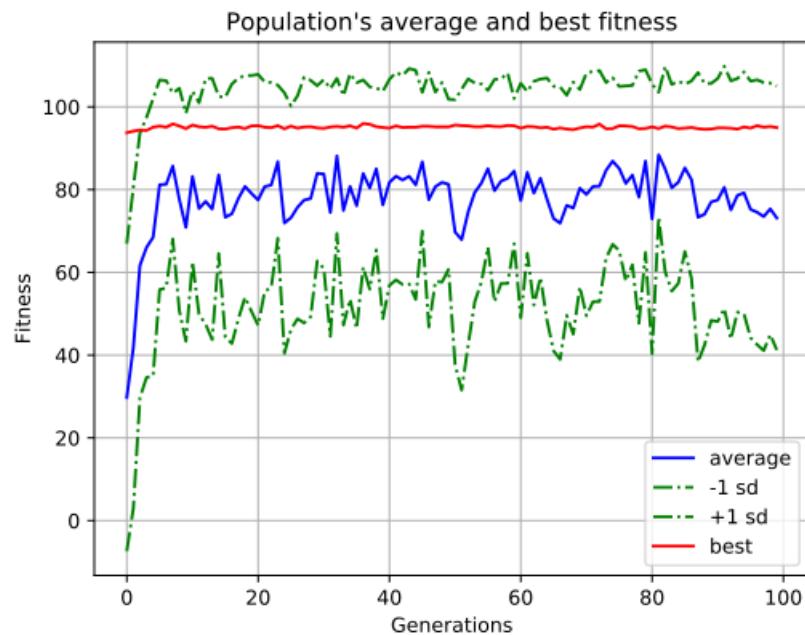


Figura 35: Evolução do *fitness* especialista inimigo 3 com população de tamanho 50.

Fonte: O autor

5.3.3.4 Resultados com a população de tamanho 100

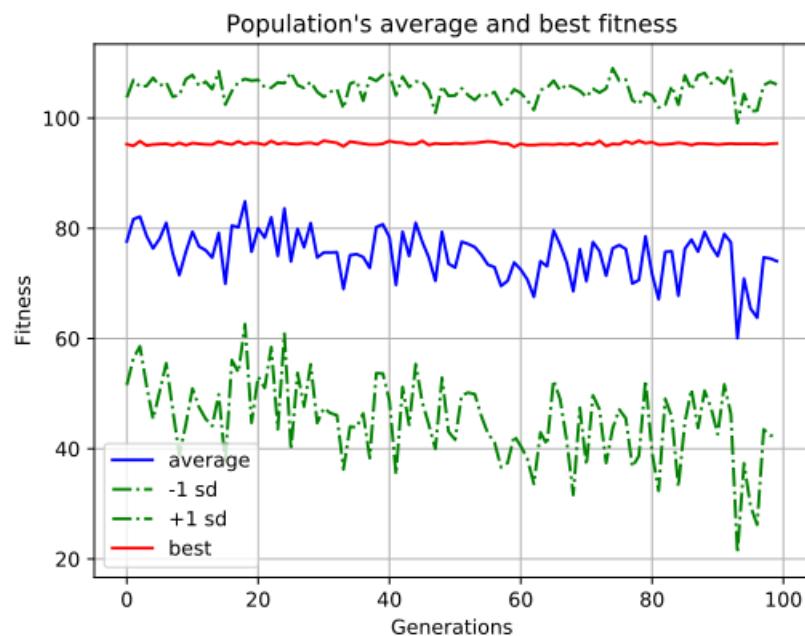


Figura 36: Evolução do *fitness* especialista inimigo 3 com população de tamanho 100.

Fonte: O autor

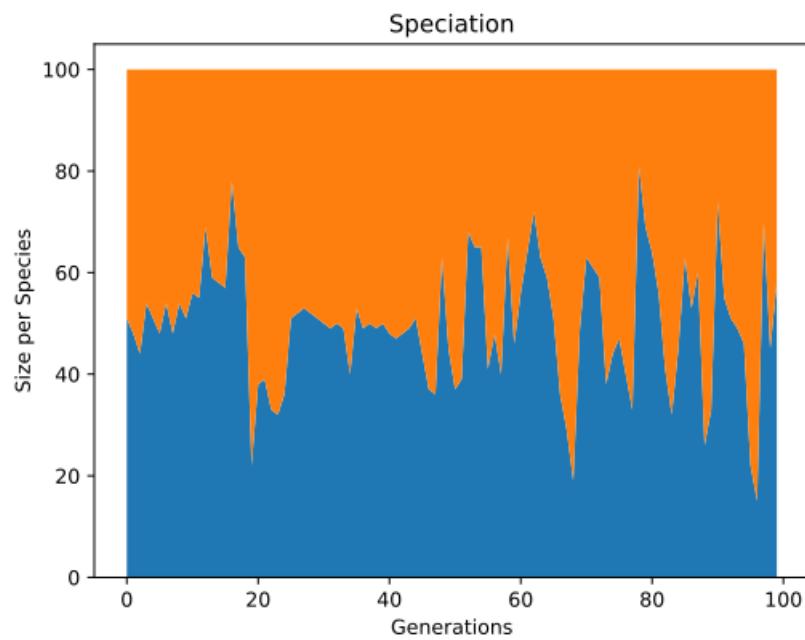


Figura 37: Variedade de espécies do *fitness* especialista inimigo 3 com população de tamanho 100.

Fonte: O autor

5.3.3.5 Resultados com a população de tamanho 150

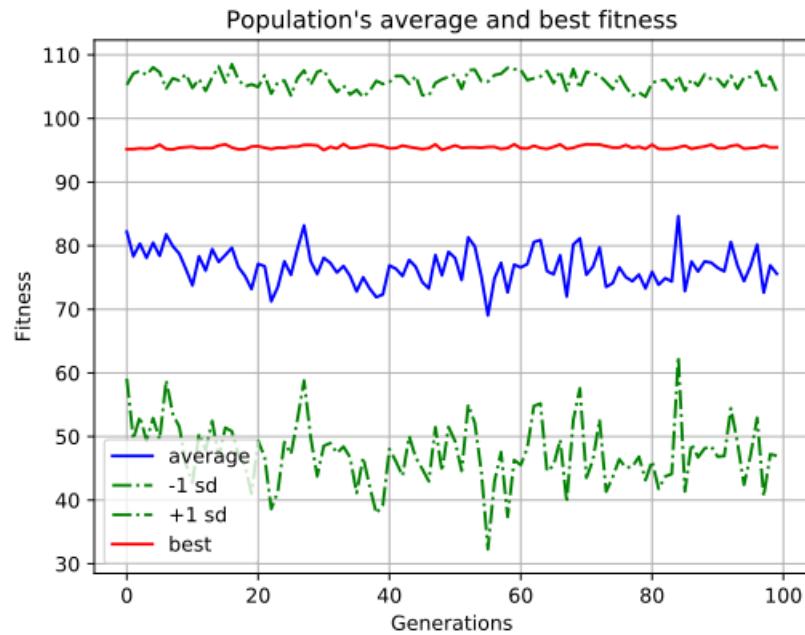


Figura 38: Evolução do *fitness* especialista inimigo 3 com população de tamanho 150.

Fonte: O autor

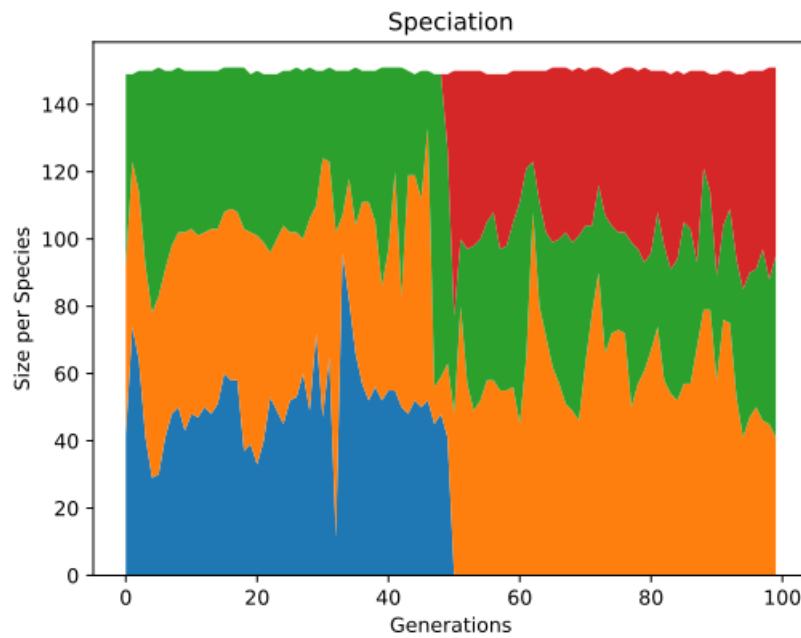


Figura 39: Variedade de espécies do *fitness* especialista inimigo 3 com população de tamanho 150.

Fonte: O autor

5.3.4 Especialista inimigo 4



Figura 40: Inimigo e cenário utilizado para o agente especialista 4

Fonte: Miras [2016]

Os resultados do especialista 4 são exibidos nas próximas Figuras 41, 42, 43, 44 e 46. Somente os testes com população de 100 e 150 indivíduos geraram mais de uma espécie no decorrer das gerações e podem ser observados nas Figuras 45 e 47.

5.3.4.1 Resultados com a população de tamanho 10

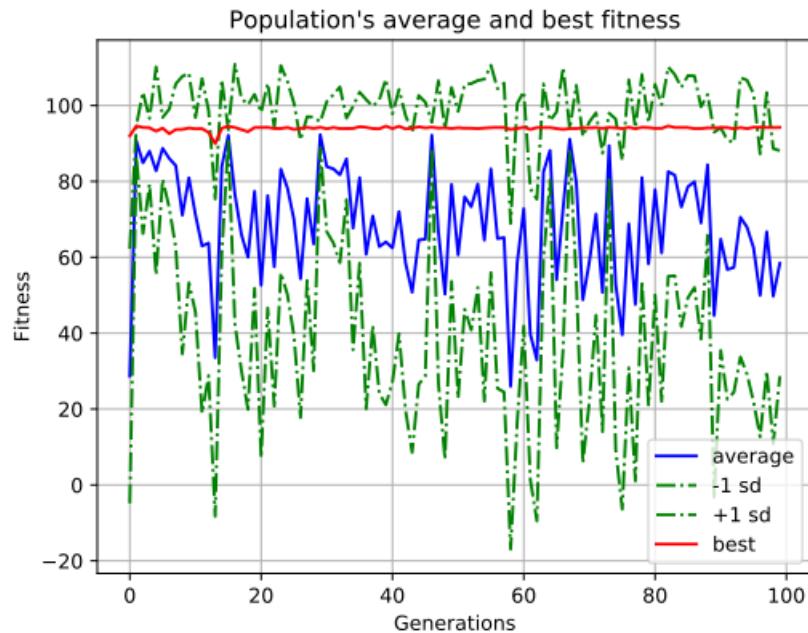


Figura 41: Evolução do *fitness* especialista inimigo 4 com população de tamanho 10.

Fonte: O autor

5.3.4.2 Resultados com a população de tamanho 25

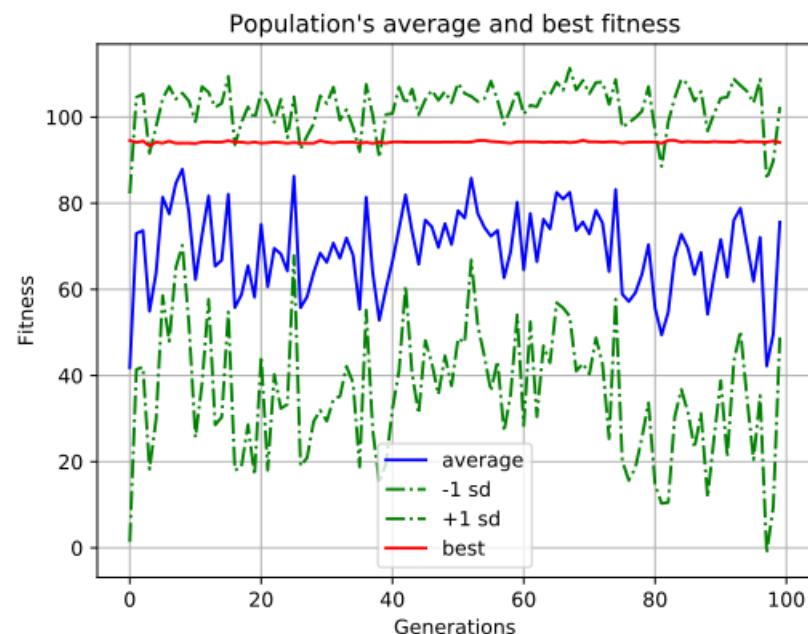


Figura 42: Evolução do *fitness* especialista inimigo 4 com população de tamanho 25.

Fonte: O autor

5.3.4.3 Resultados com a população de tamanho 50

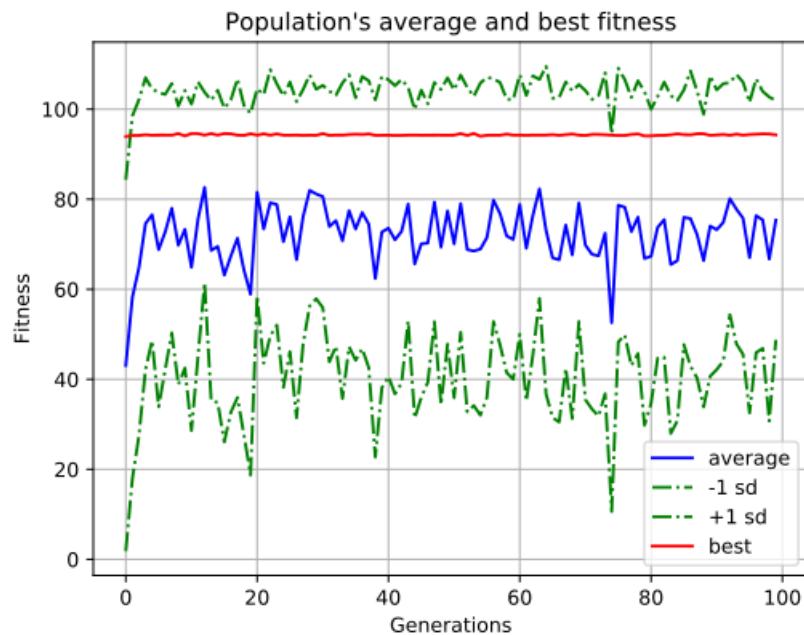


Figura 43: Evolução do *fitness* especialista inimigo 4 com população de tamanho 50.

Fonte: O autor

5.3.4.4 Resultados com a população de tamanho 100

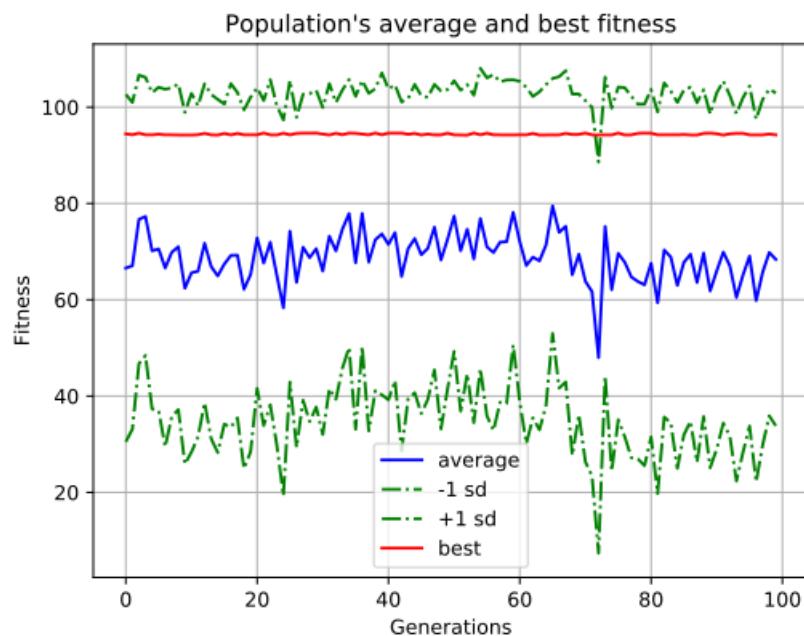


Figura 44: Evolução do *fitness* especialista inimigo 4 com população de tamanho 100.

Fonte: O autor

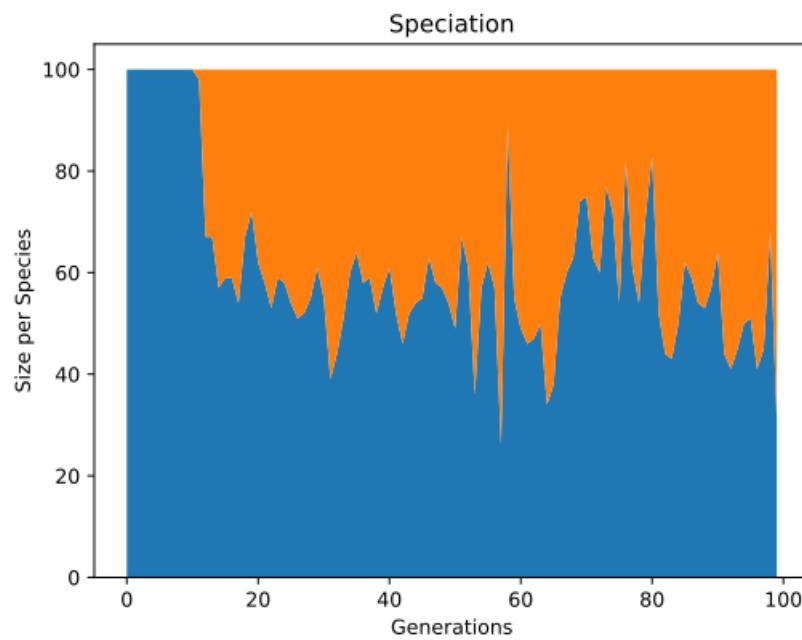


Figura 45: Variedade de espécies do *fitness* especialista inimigo 4 com população de tamanho 100.

Fonte: O autor

5.3.4.5 Resultados com a população de tamanho 150

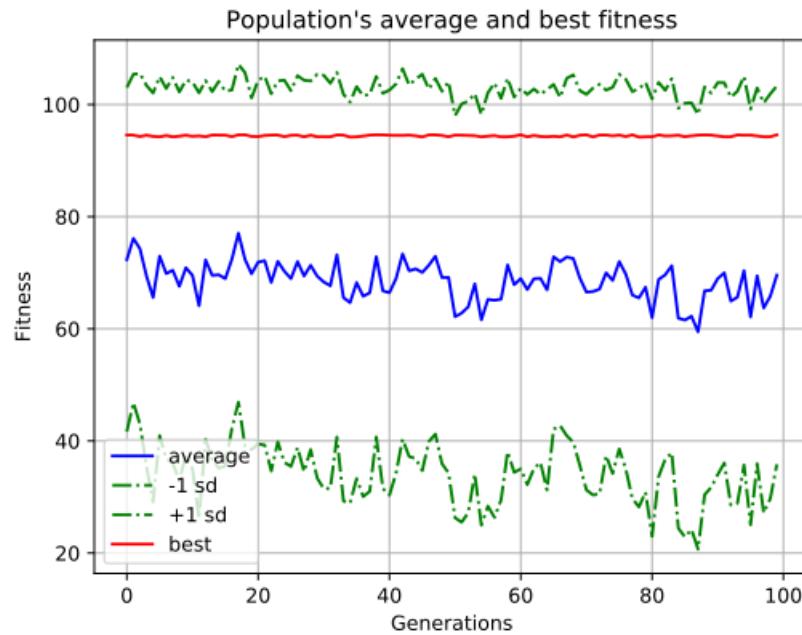


Figura 46: Evolução do *fitness* especialista inimigo 4 com população de tamanho 150.

Fonte: O autor

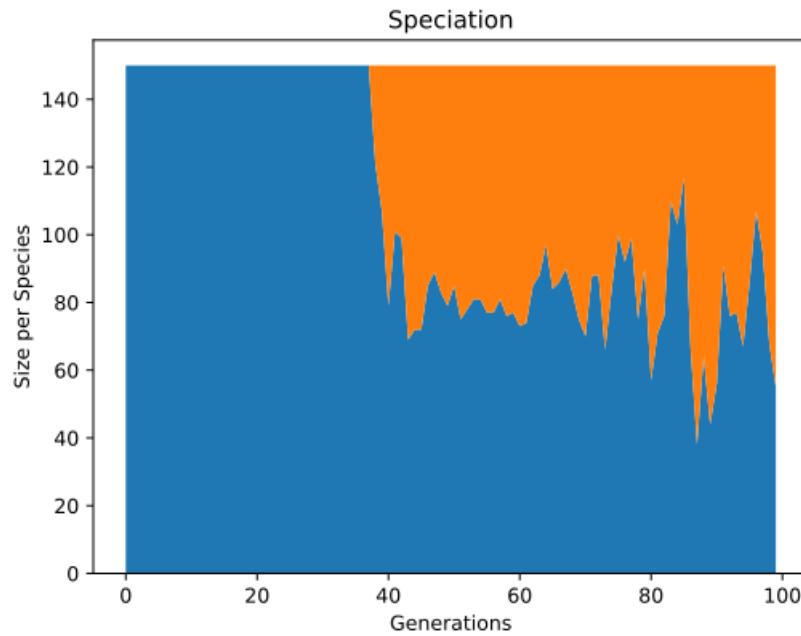


Figura 47: Variedade de espécies do *fitness* especialista inimigo 4 com população de tamanho 150.

Fonte: O autor

5.3.5 Especialista inimigo 5



Figura 48: Inimigo e cenário utilizado para o agente especialista 5

Fonte: Miras [2016]

Os resultados do especialista 5 são exibidos nas próximas Figuras 49, 50, 51, 52 e 54. Somente os testes com população de 100 e 150 indivíduos geraram mais de uma espécie no decorrer das gerações e podem ser observados nas Figuras 53 e 55.

5.3.5.1 Resultados com a população de tamanho 10

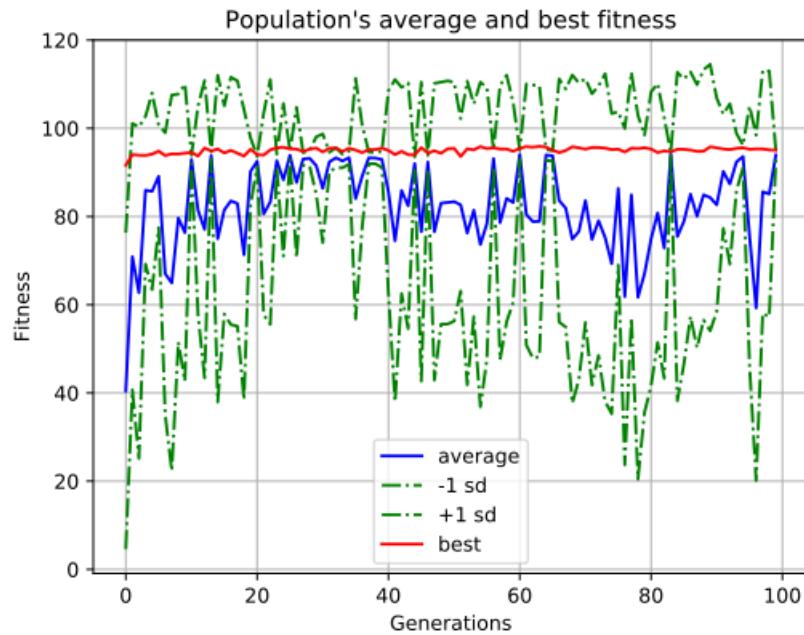


Figura 49: Evolução do *fitness* especialista inimigo 5 com população de tamanho 10.

Fonte: O autor

5.3.5.2 Resultados com a população de tamanho 25

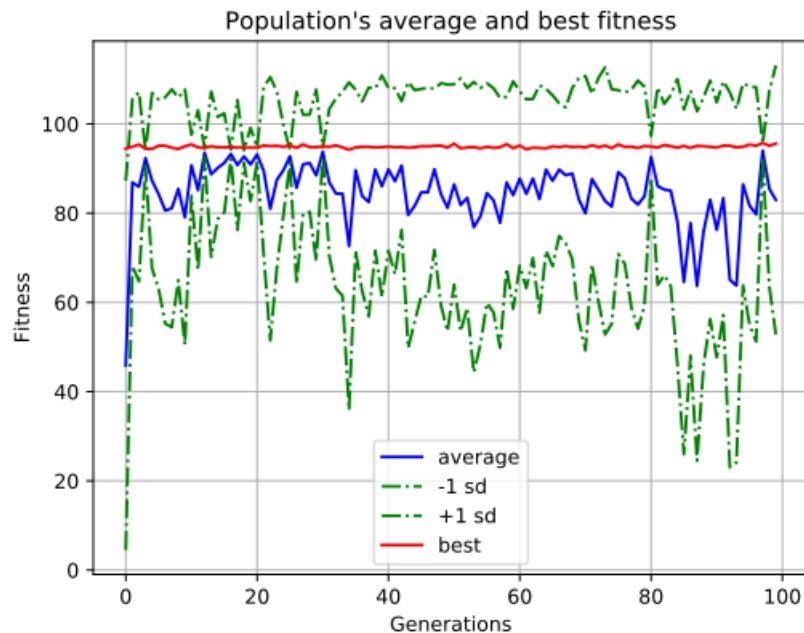


Figura 50: Evolução do *fitness* especialista inimigo 5 com população de tamanho 25.

Fonte: O autor

5.3.5.3 Resultados com a população de tamanho 50

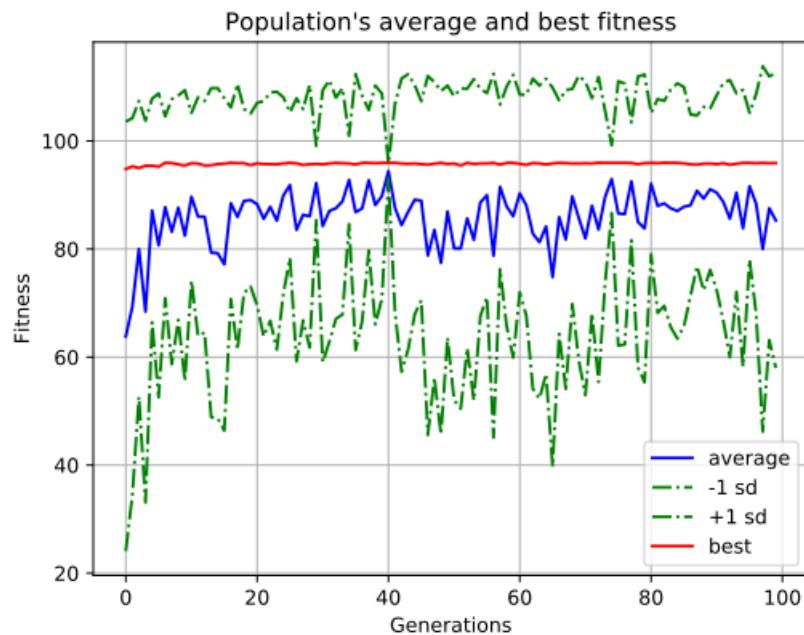


Figura 51: Evolução do *fitness* especialista inimigo 5 com população de tamanho 50.

Fonte: O autor

5.3.5.4 Resultados com a população de tamanho 100

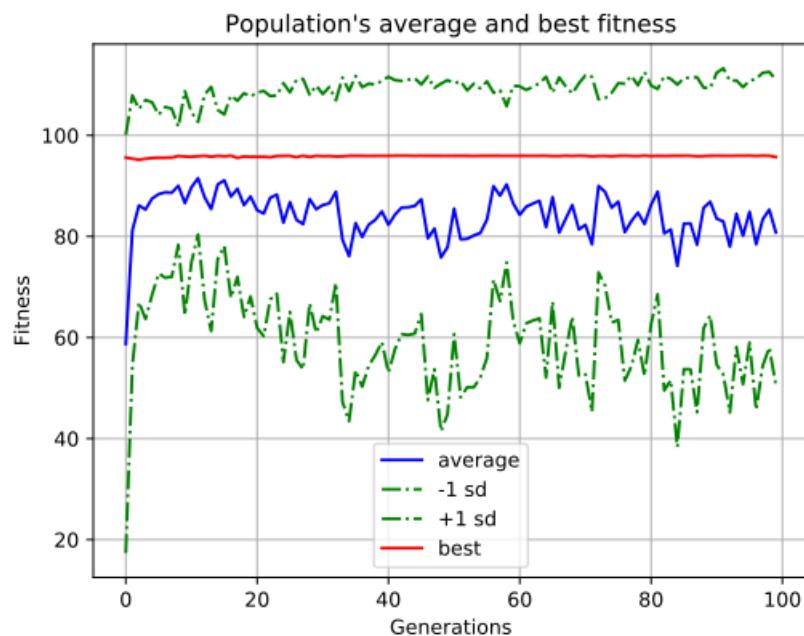


Figura 52: Evolução do *fitness* especialista inimigo 5 com população de tamanho 100.

Fonte: O autor

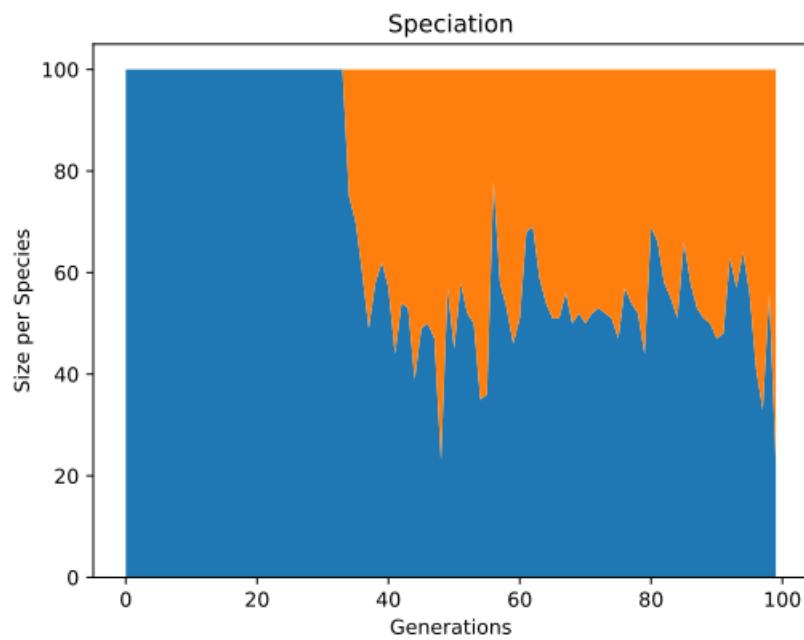


Figura 53: Variedade de espécies do *fitness* especialista inimigo 5 com população de tamanho 100.

Fonte: O autor

5.3.5.5 Resultados com a população de tamanho 150

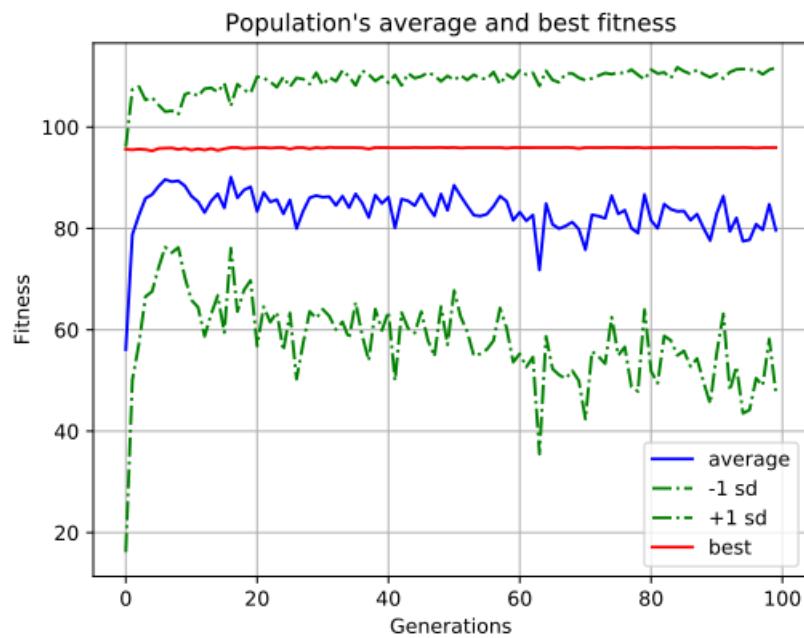


Figura 54: Evolução do *fitness* especialista inimigo 5 com população de tamanho 150.

Fonte: O autor

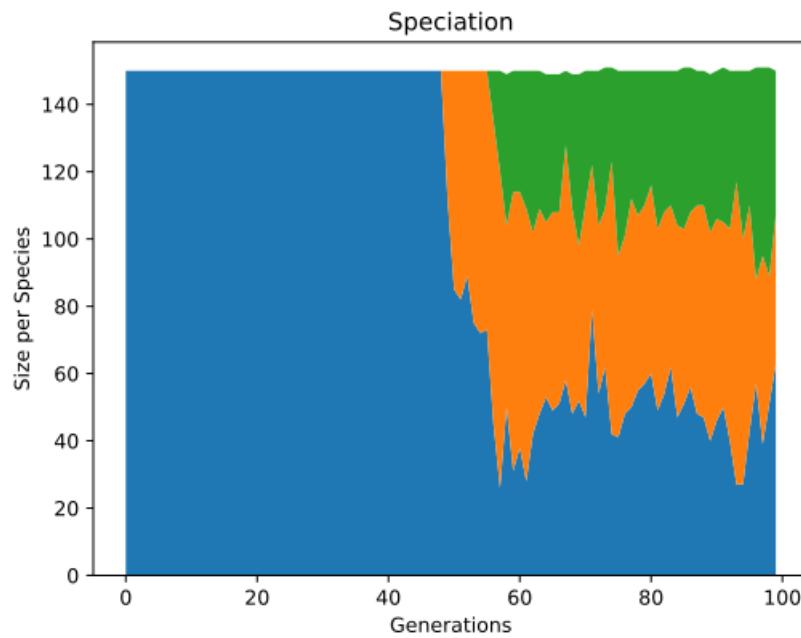


Figura 55: Variedade de espécies do *fitness* especialista inimigo 5 com população de tamanho 150.

Fonte: O autor

5.3.6 Especialista inimigo 6



Figura 56: Inimigo e cenário utilizado para o agente especialista 6

Fonte: Miras [2016]

Os resultados do especialista 6 são exibidos nas próximas Figuras 57, 58, 59, 60 e 61. Somente o teste com população de 150 indivíduos gerou mais de uma espécie no decorrer das gerações e pode ser observado na Figura 62.

5.3.6.1 Resultados com a população de tamanho 10

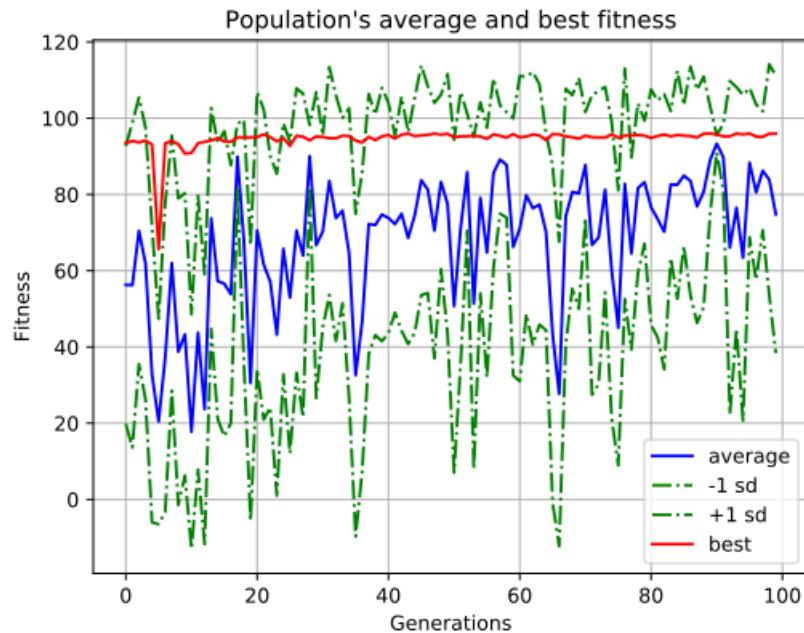


Figura 57: Evolução do *fitness* especialista inimigo 6 com população de tamanho 10.

Fonte: O autor

5.3.6.2 Resultados com a população de tamanho 25

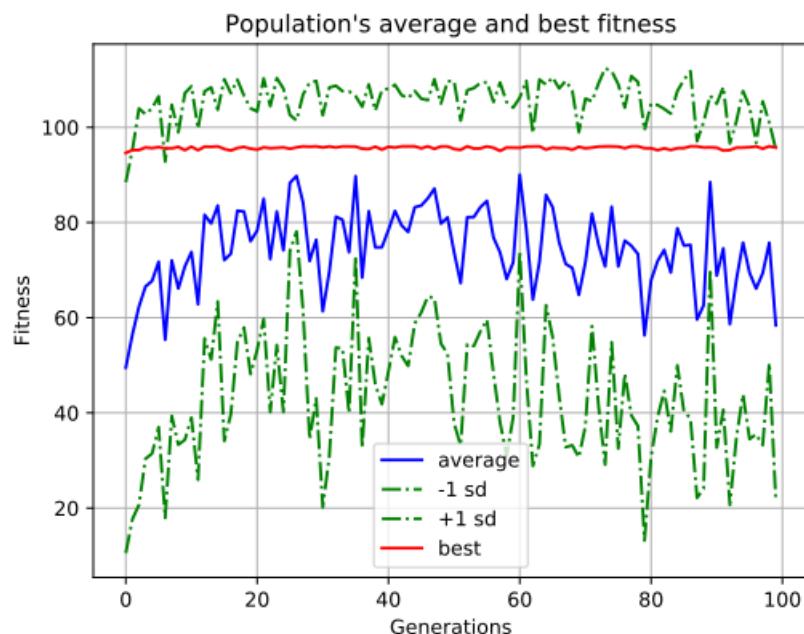


Figura 58: Evolução do *fitness* especialista inimigo 6 com população de tamanho 25.

Fonte: O autor

5.3.6.3 Resultados com a população de tamanho 50

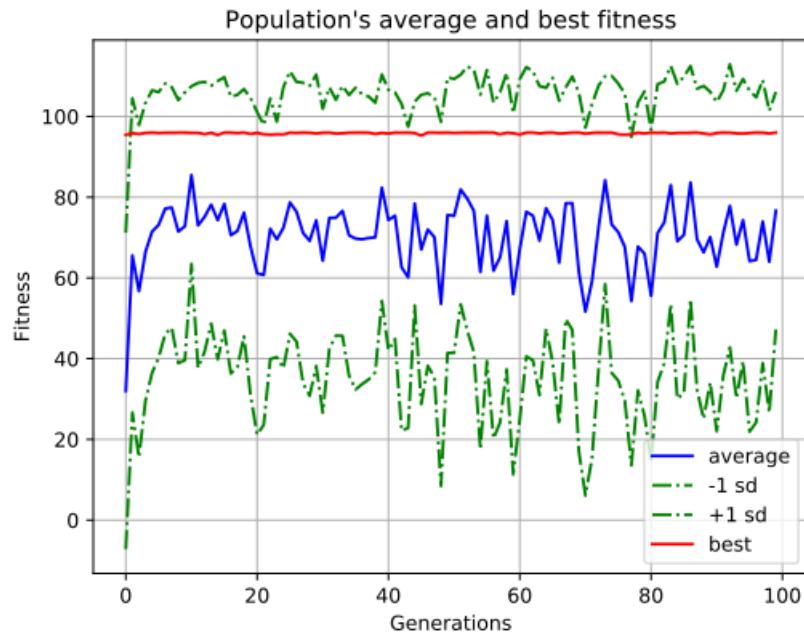


Figura 59: Evolução do *fitness* especialista inimigo 6 com população de tamanho 50.

Fonte: O autor

5.3.6.4 Resultados com a população de tamanho 100

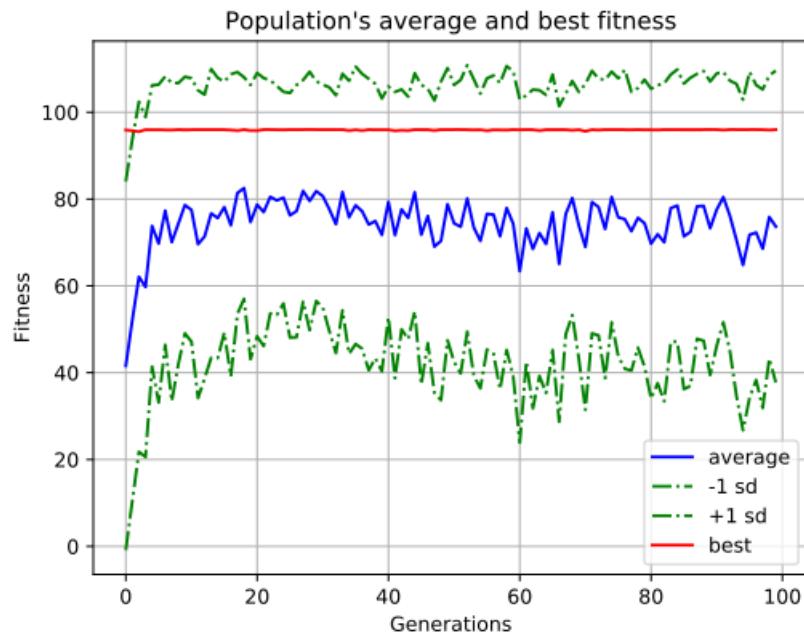


Figura 60: Evolução do *fitness* especialista inimigo 6 com população de tamanho 100.

Fonte: O autor

5.3.6.5 Resultados com a população de tamanho 150

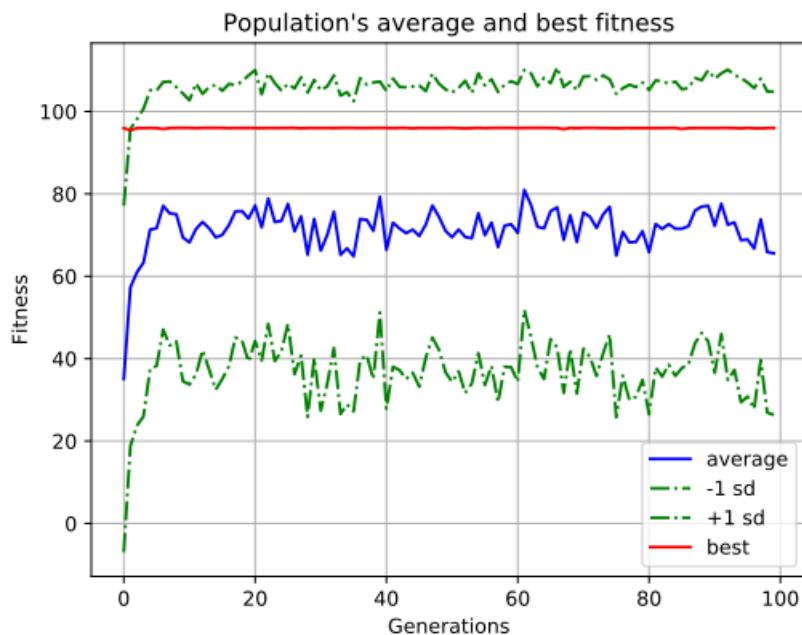


Figura 61: Evolução do *fitness* especialista inimigo 6 com população de tamanho 150.

Fonte: O autor

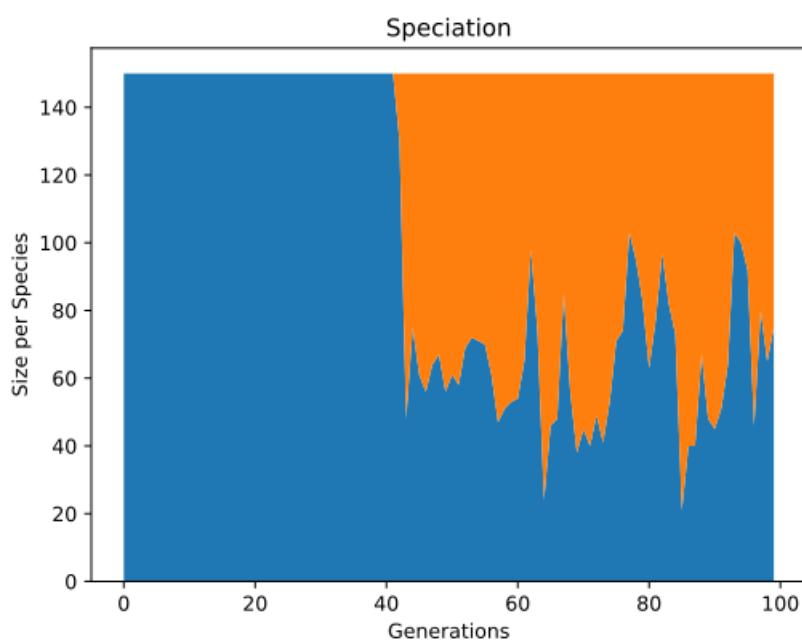


Figura 62: Variedade de espécies do *fitness* especialista inimigo 6 com população de tamanho 150.

Fonte: O autor

5.3.7 Especialista inimigo 7

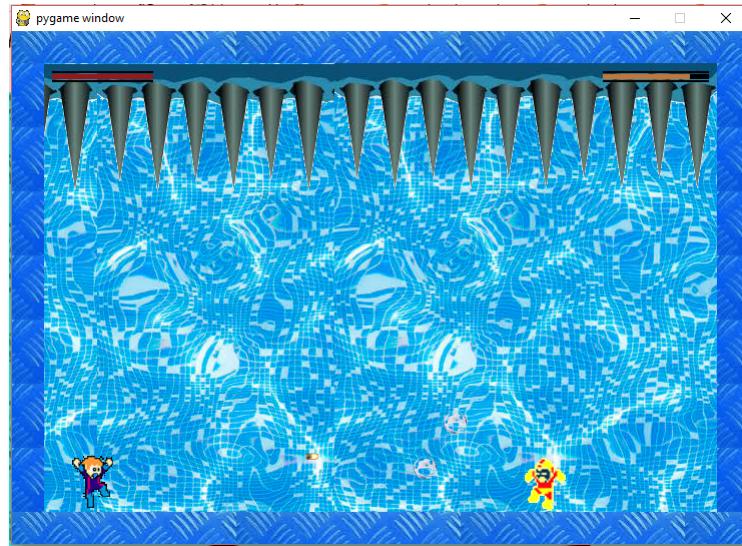


Figura 63: Inimigo e cenário utilizado para o agente especialista 7

Fonte: Miras [2016]

Os resultados do especialista 7 são exibidos nas próximas Figuras 64, 65, 66, 67 e 69. Somente os testes com população de 100 e 150 indivíduos geraram mais de uma espécie no decorrer das gerações e podem ser observados nas Figuras 68 e 70.

5.3.7.1 Resultados com a população de tamanho 10

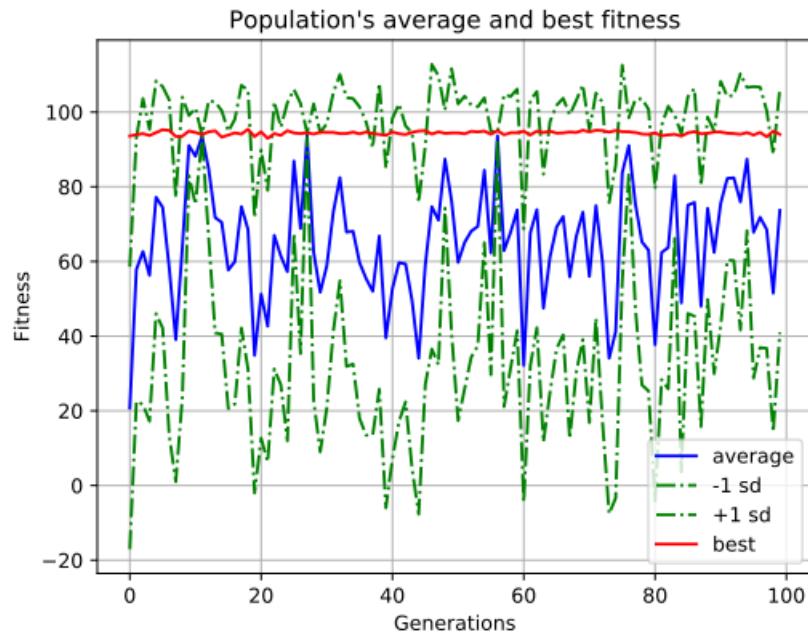


Figura 64: Evolução do *fitness* especialista inimigo 7 com população de tamanho 10.

Fonte: O autor

5.3.7.2 Resultados com a população de tamanho 25

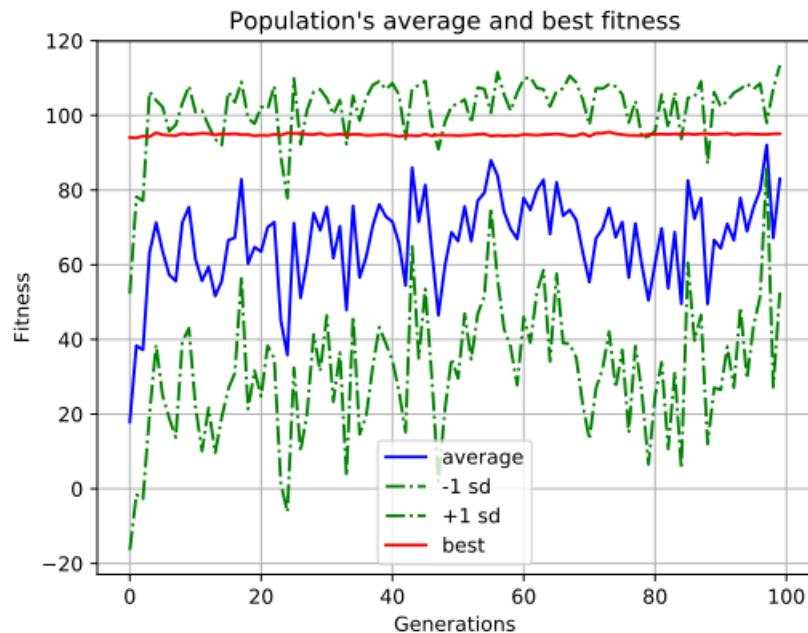


Figura 65: Evolução do *fitness* especialista inimigo 7 com população de tamanho 25.

Fonte: O autor

5.3.7.3 Resultados com a população de tamanho 50

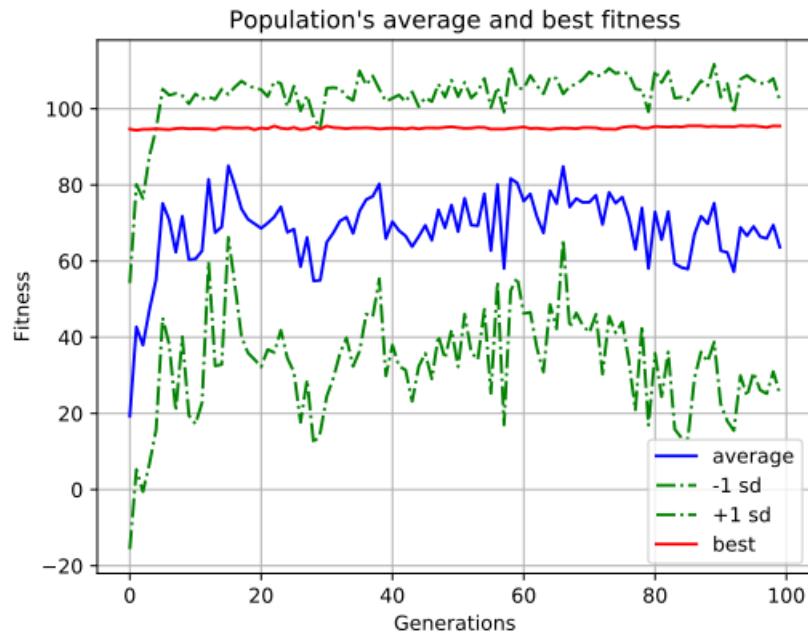


Figura 66: Evolução do *fitness* especialista inimigo 7 com população de tamanho 50.

Fonte: O autor

5.3.7.4 Resultados com a população de tamanho 100

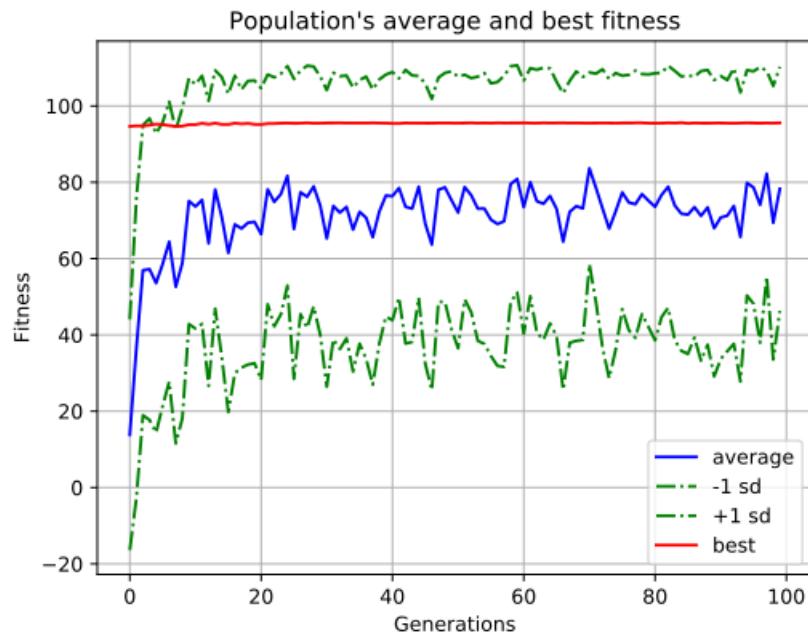


Figura 67: Evolução do *fitness* especialista inimigo 7 com população de tamanho 100.

Fonte: O autor

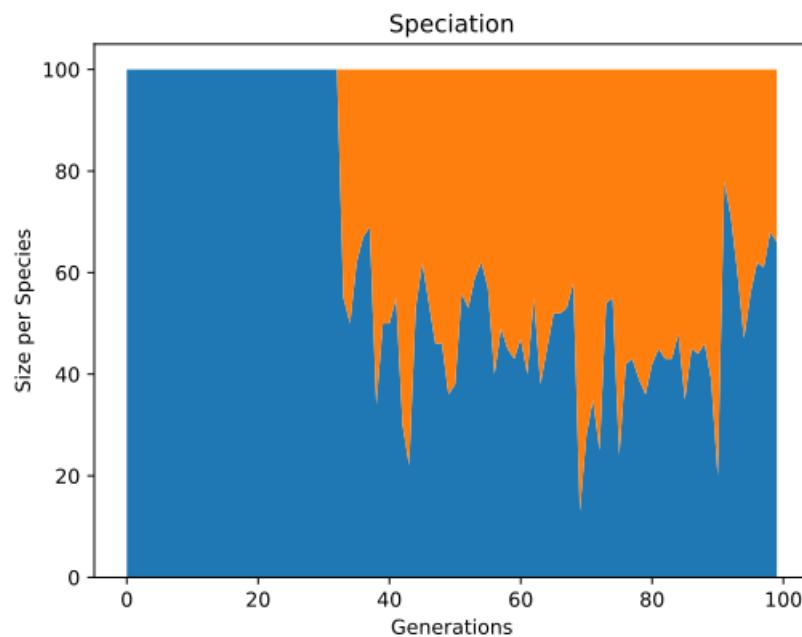


Figura 68: Variedade de espécies do *fitness* especialista inimigo 7 com população de tamanho 100.

Fonte: O autor

5.3.7.5 Resultados com a população de tamanho 150

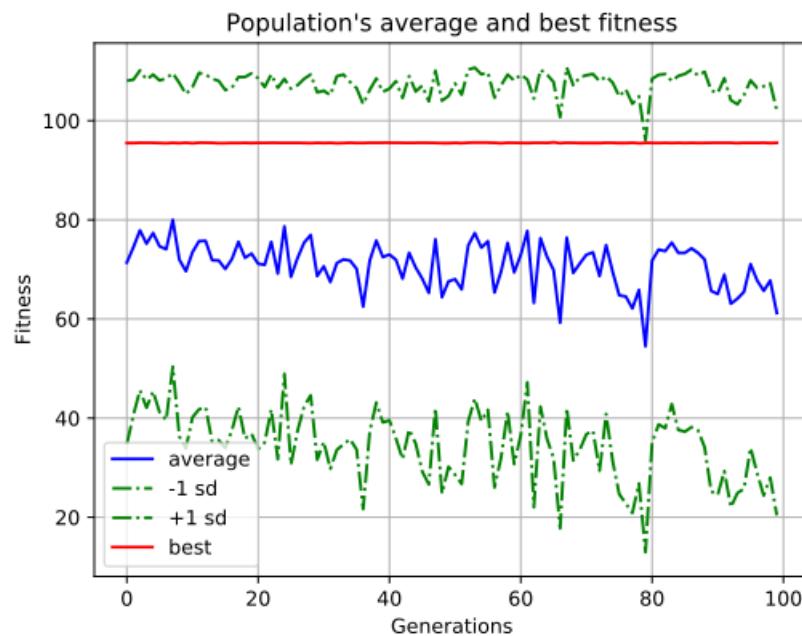


Figura 69: Evolução do *fitness* especialista inimigo 7 com população de tamanho 150.

Fonte: O autor

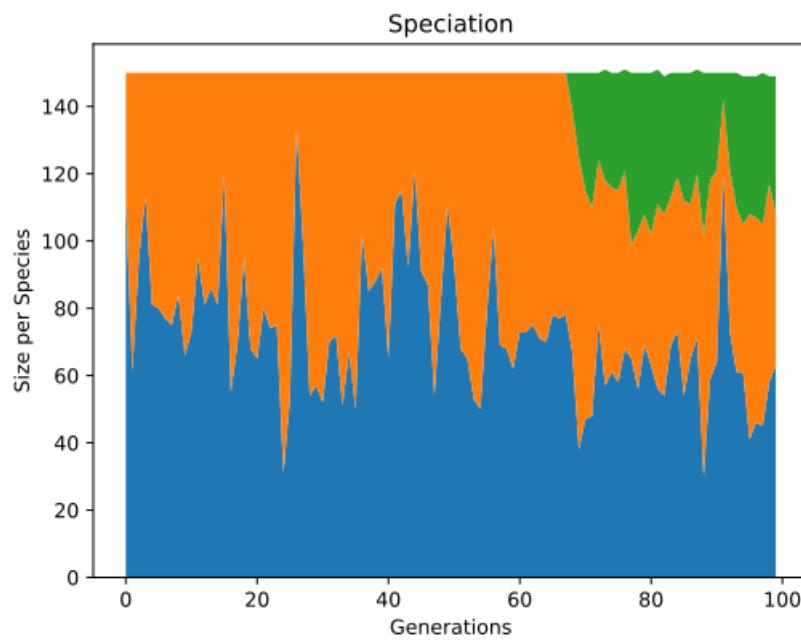


Figura 70: Variedade de espécies do *fitness* especialista inimigo 7 com população de tamanho 150.

Fonte: O autor

5.3.8 Especialista inimigo 8

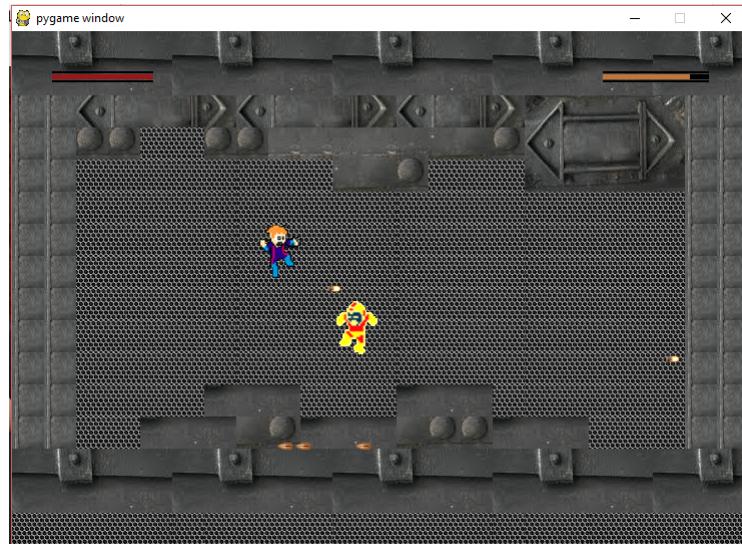


Figura 71: Inimigo e cenário utilizado para o agente especialista 8

Fonte: Miras [2016]

Os resultados do especialista 8 são exibidos nas próximas Figuras 72, 73, 74, 75 e 77. Somente os testes com população de 100 e 150 indivíduos geraram mais de uma espécie no decorrer das gerações e podem ser observados nas Figuras 76 e 78.

5.3.8.1 Resultados com a população de tamanho 10

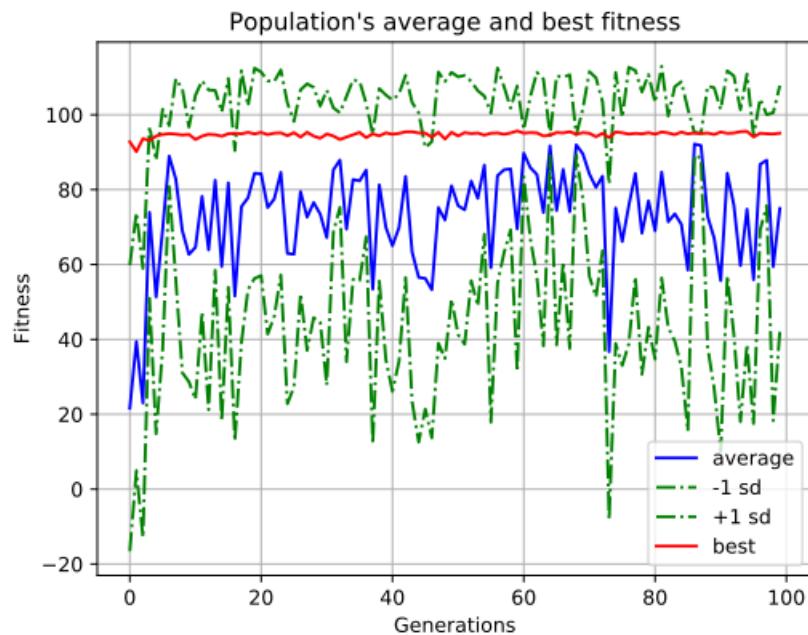


Figura 72: Evolução do *fitness* especialista inimigo 8 com população de tamanho 10.

Fonte: O autor

5.3.8.2 Resultados com a população de tamanho 25

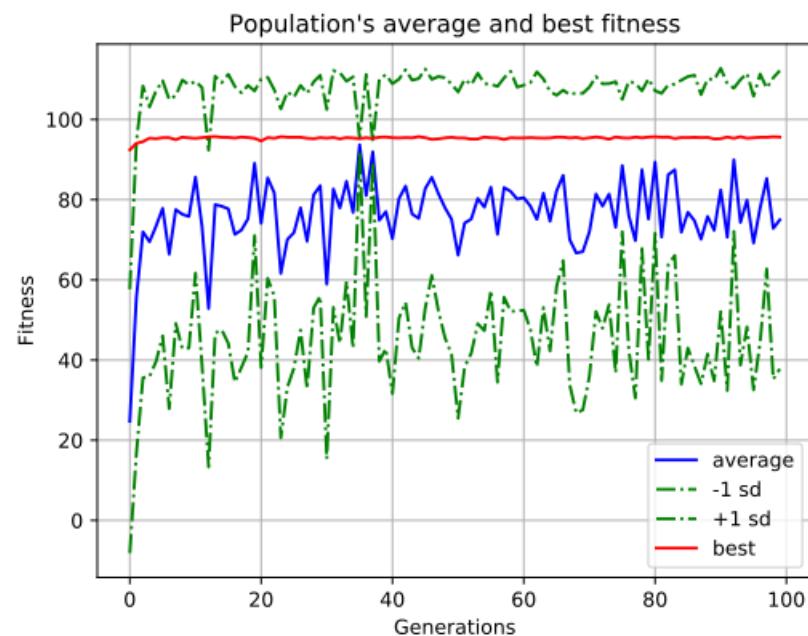


Figura 73: Evolução do *fitness* especialista inimigo 8 com população de tamanho 25.

Fonte: O autor

5.3.8.3 Resultados com a população de tamanho 50

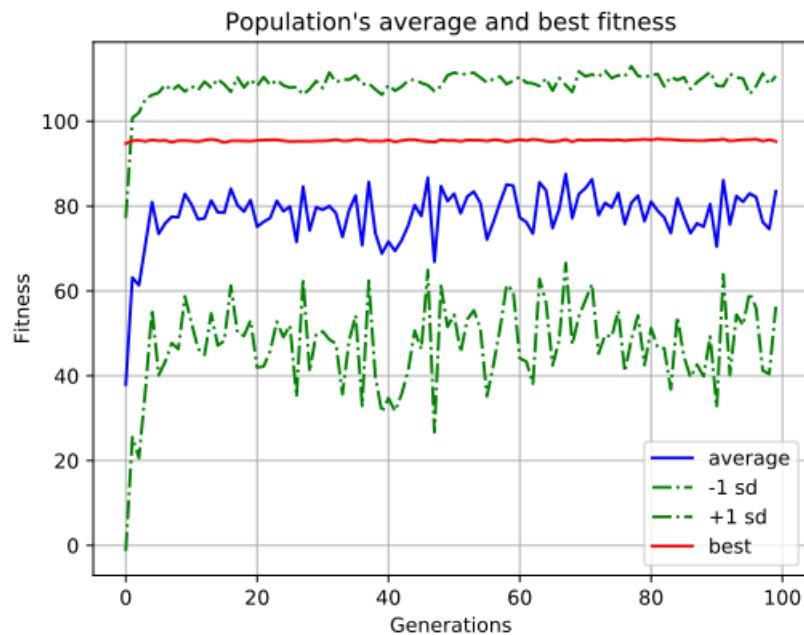


Figura 74: Evolução do *fitness* especialista inimigo 8 com população de tamanho 50.

Fonte: O autor

5.3.8.4 Resultados com a população de tamanho 100

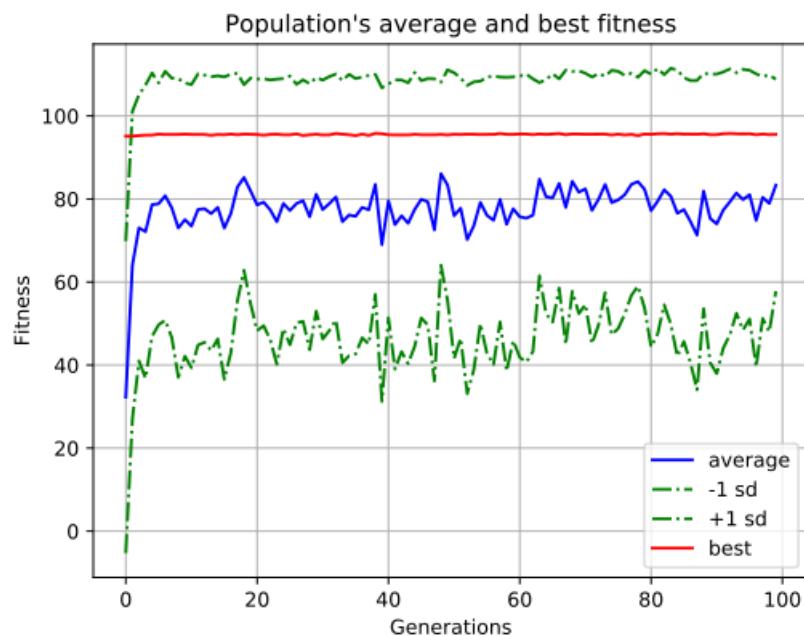


Figura 75: Evolução do *fitness* especialista inimigo 8 com população de tamanho 100.

Fonte: O autor

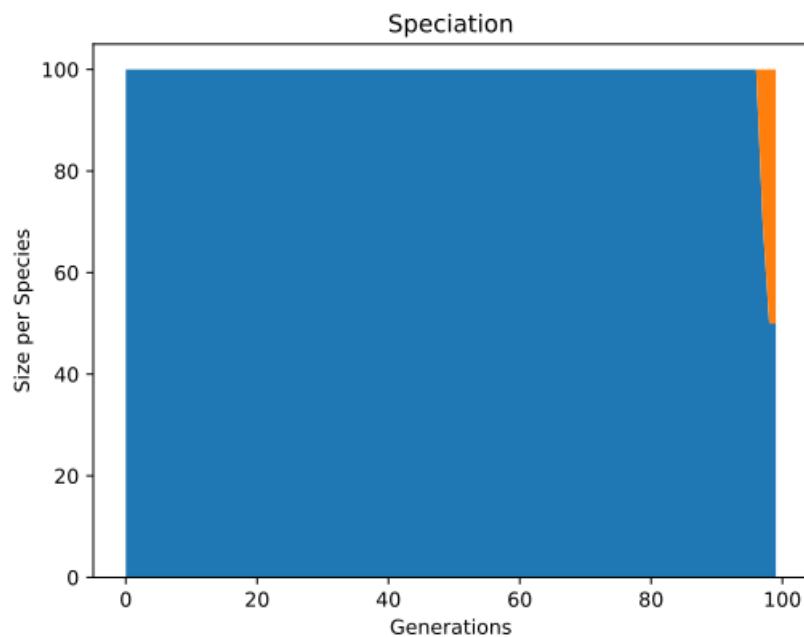


Figura 76: Variedade de espécies do *fitness* especialista inimigo 8 com população de tamanho 100.

Fonte: O autor

5.3.8.5 Resultados com a população de tamanho 150

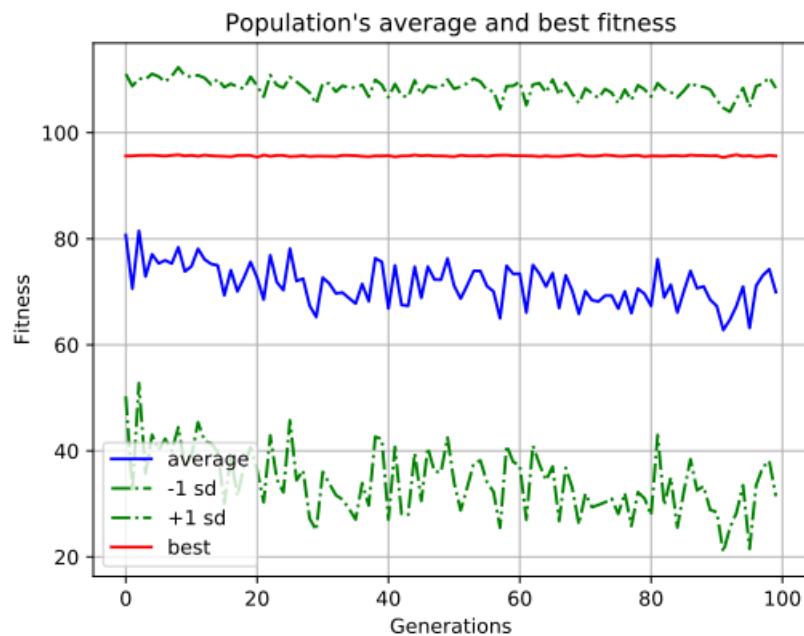


Figura 77: Evolução do *fitness* especialista inimigo 8 com população de tamanho 150.

Fonte: O autor

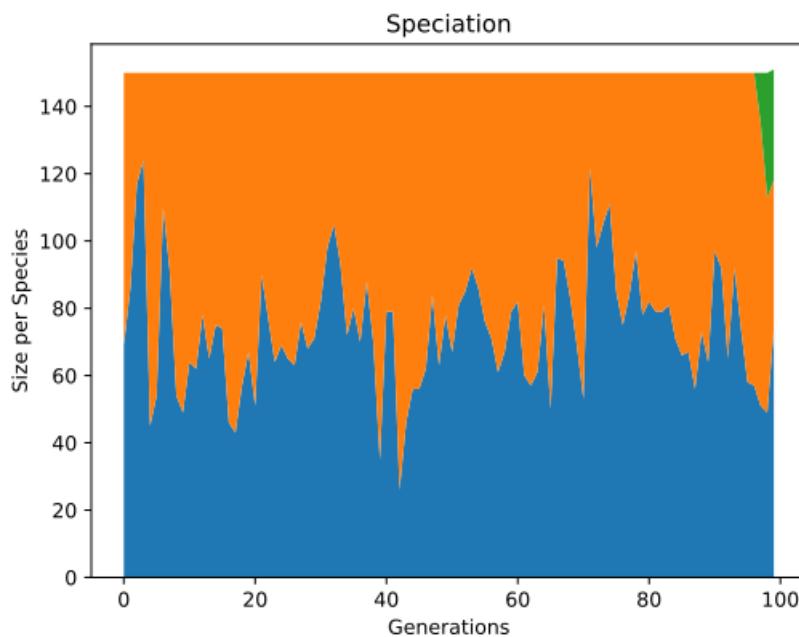


Figura 78: Variedade de espécies do *fitness* especialista inimigo 8 com população de tamanho 150.

Fonte: O autor

5.4 Considerações finais

O objetivo deste capítulo foi a apresentação da solução desenvolvida, testes realizados, tecnologias envolvidas e resultados. Os tópicos abordados foram os seguintes:

- A lógica do algoritmo implementado;
- A linguagem de programação utilizada e suas características;
- A biblioteca chave utilizada para a implementação do algoritmo;
- Explicação sobre trechos de código mais relevantes para a implementação;
- Testes e resultados.

A apresentação dos resultados levantaram um ponto interessante com relação à divisão de espécies que será discutido no capítulo 6, que terá como objetivo a finalização e discussão sobre os resultados obtidos no desenvolvimento desta monografia.

Outro ponto interessante foi a utilização da biblioteca NEAT-Python para a manipulação da população e processo evolutivo, facilitando o processo de implementação da solução.

6 Conclusões e trabalhos futuros

Esta monografia teve como objetivo o estudo e desenvolvimento da aplicação de um algoritmo neuro-evolutivo à mecânica de um jogo eletrônico 2D, ou seja, um agente autônomo capaz de controlar a mecânica de um personagem e cumprir um determinado objetivo.

A complexidade da mecânica de um jogo eletrônico 2D, juntamente com o desafio de dominar essa mecânica e derrotar um oponente torna o processo de aprendizagem de um agente autônomo com esse objetivo um cenário bom para a aplicação da neuro-evolução.

A técnica de neuro-evolução escolhida para o desenvolvimento da solução foi o NEAT (*neuroevolution of augmenting topologies*), por possibilitar a evolução da estrutura da rede neural como um todo [STANLEY; MIIKKULAINEN, 2002]. Para a mecânica de jogo eletrônico 2D foi escolhido o *framework* EvoMan, que fornece 8 jogos de presa-predador inspirados no jogo eletrônico *MegaMan II*, capaz de simular um ambiente dinâmico, ou seja, que não apresenta sempre o mesmo comportamento [MIRAS, 2016]. O algoritmo implementado obteve sucesso no controle da mecânica do personagem, sendo apresentado os resultados para diferentes configurações de ambiente e parâmetros do algoritmo.

Pontos relevantes levantados durante o processo de estudo e desenvolvimento sugeriram, como por exemplo a observação de que a diversificação de espécies aumentou somente quando a população era igual ou maior que 100 genomas. Um teste futuro interessante a ser realizado é colocar populações de tamanho ainda maiores que os testados.

Outro trabalho futuro relevante é o teste de um agente generalista como explicado no capítulo 4 para observação do comportamento do algoritmo para a evolução de uma rede neural com multi-objetivos.

Um teste relevante para ser realizado é a alteração do atributo *weight_max_value* para o valor de 1 ao invés de 30 e o *weight_min_value* para 0 ao invés de -30. Essa distância grande nos pesos pode ter influenciado no crescimento baixo do *fitness* no decorrer das gerações. Alterando o valor desses atributos e refazendo todos os 40 testes comparando os resultados com os obtidos nesta monografia poderiam confirmar essa hipótese.

A implementação completa do algoritmo pode ser encontrada e utilizada neste link da plataforma GitHub: <https://github.com/filipecavalc/Algoritmo-neuro-evolutivo-aplicado-a-mecanica-de-um-jogo-2D>

Referências

- AHA, D. W. *Lazy learning*. [S.l.]: Norwell, 1997. Citado na página 7.
- AHA, D. W.; KIBLER, D.; ALBERT, M. K. Instance-based learning algorithms. *Machine Learning*, v. 6, 1991. Citado na página 7.
- ANDERSEN, T.; STANLEY, K. O.; MIIKKULAINEN, R. *Neuro-Evolution Through Augmenting Topologies Applied To Evolving Neural Networks To Play Othello*. 2002. Disponível em: [⟨ftp://ftp.cs.utexas.edu/pub/techreports/tr02-32.pdf⟩](ftp://ftp.cs.utexas.edu/pub/techreports/tr02-32.pdf). Acesso em: 10 de setembro de 2017. Citado na página 18.
- ARAUJO, K. d. S. M. de; FRANCA, F. O. de. Evolving a generalized strategy for an action-platformer video game framework. In: IEEE. *Evolutionary Computation (CEC), 2016 IEEE Congress on*. [S.l.], 2016. p. 1303–1310. Citado 4 vezes nas páginas 4, 1, 11 e 12.
- BARANAUSKAS, J. A. *Aprendizado de Máquina Conceitos e Definições*. 2007. Disponível em: [⟨http://dcm.ffclrp.usp.br/~augusto/teaching/ami/AM-I-Conceitos-Definicoes.pdf⟩](http://dcm.ffclrp.usp.br/~augusto/teaching/ami/AM-I-Conceitos-Definicoes.pdf). Acesso em: 02 de Setembro 2017. Citado 3 vezes nas páginas 3, 4 e 6.
- BLING, S. *NEATEvolve.lua*. 2015. Disponível em: [⟨https://pastebin.com/ZZmSNaHX⟩](https://pastebin.com/ZZmSNaHX). Acesso em: 10 de setembro de 2017. Citado na página 19.
- CALIXTO, W. et al. Desenvolvimento de operador matemático para algoritmos de otimização heurísticos aplicado a problema de geoprospecção. *TEMA*, v. 15, 2014. Citado na página 9.
- CRUZ, A. A. da. *Algoritmos evolutivos com inspiração quântica para problemas com representação numérica*. Tese (Doutorado) — PUC-Rio, 2007. Citado na página 12.
- ENGELBRECHT, A. P. *Computacional Intelligence An introduction Second Edition*. [S.l.]: Wiley, 2007. Citado na página 8.
- GOLDBERG, D. E. *Genetic Algorithms in Search, Optimization and Machine Learning*. [S.l.]: Addison-Wesley Longman, 1989. Citado na página 8.
- HAUSKNECHT, M. et al. A neuroevolution approach to general atari game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, IEEE, v. 6, n. 4, p. 355–366, 2014. Citado na página 21.
- HEINEN, M. R.; OSÓRIO, F. S. *Autenticação de assinaturas utilizando algoritmos de Aprendizado de Máquina*. 2005. Disponível em: [⟨http://osorio.wait4.org/publications/heinen-osorio-enia2005.pdf⟩](http://osorio.wait4.org/publications/heinen-osorio-enia2005.pdf). Acesso em: 01 de Setembro de 2017. Citado na página 3.
- HOSCH, W. L. *machine learning*. 2009. Disponível em: [⟨https://global.britannica.com/technology/machine-learning⟩](https://global.britannica.com/technology/machine-learning). Acesso em: 31 de Agosto de 2017. Citado 2 vezes nas páginas 4 e 2.

- KCTS. *The Video Game Revolution*. 2015. Disponível em: [\(<http://www.pbs.org/kcts/videogamerevolution/history/timeline_flash.html>\)](http://www.pbs.org/kcts/videogamerevolution/history/timeline_flash.html). Acesso em: 22 de novembro de 2017. Citado na página 1.
- LAMARCK, J.-B. *Texto de uma palestra dada por Lamarck no Musée National d'Histoire Naturelle*. 1803. Disponível em: [\(<http://www.ucmp.berkeley.edu/history/lamarck.html>\)](http://www.ucmp.berkeley.edu/history/lamarck.html). Acesso em: 22 de novembro de 2017. Citado na página 16.
- LEHMAN, J.; MIIKKULAINEN, R. Neuroevolution. *Scholarpedia*, v. 8, n. 6, p. 30977, 2013. Revision #133684. Citado 9 vezes nas páginas 4, 1, 12, 13, 14, 15, 16, 17 e 18.
- MICHALSKI, R. S.; BRATKO, I.; KUBAT, M. *Machine Learning and Data Mining: Methods and Applications*. [S.l.]: Wiley, 1998. Citado na página 3.
- MIRAS, K. *EvoMan - Framework 1.0*. 2016. Disponível em: [\(<https://github.com/karinemiras/evoman_framework/blob/master/evoman1.0-doc.pdf>\)](https://github.com/karinemiras/evoman_framework/blob/master/evoman1.0-doc.pdf). Acesso em: 7 de novembro de 2017. Citado 17 vezes nas páginas 1, 23, 24, 25, 26, 27, 28, 29, 41, 45, 49, 53, 57, 61, 65, 69 e 74.
- MONARD, M. C.; BARANAUSKAS, J. A. *Conceitos sobre Aprendizado de Maquina*. 2003. Disponível em: [\(<http://dcm.ffclrp.usp.br/~augusto/publications/2003-sistemas-inteligentes-cap4.pdf>\)](http://dcm.ffclrp.usp.br/~augusto/publications/2003-sistemas-inteligentes-cap4.pdf). Acesso em: 01 de Setembro de 2017. Citado 6 vezes nas páginas 3, 4, 6, 7, 8 e 9.
- PRATI, R. C.; BARANAUSKAS, J. A.; MONARD, M. C. Padronizaçao da sintaxe e informaçoes sobre regras induzidas a partir de algoritmos de aprendizado de máquina simbólico. *Revista Eletrônica de Iniciaçao Científica*, v. 2, n. 3, p. 21, 2002. Citado 2 vezes nas páginas 2 e 3.
- PYTHON SOFTWARE FOUNDATION. *Python Software Foundation*. 2003. Disponível em: [\(<https://www.python.org/psf/>\)](https://www.python.org/psf/). Acesso em: 20 de novembro de 2017. Citado na página 31.
- RODRIGUES, R. de P. *RECONHECIMENTO DE CARGAS ELÉTRICAS MONOFÁSICAS NÃO-LINEARES ATRAVÉS DA DECOMPOSIÇÃO WAVELET E DE REDES NEURAIS ARTIFICIAIS*. Itajubá - MG: [s.n.], 2009. Citado na página 8.
- ROQUE, A. C. *Modelos de Plasticidade Sináptica em Redes Neurais*. 2015. Disponível em: [\(<http://educacaosec21.org.br/wp-content/uploads/2013/08/RedeCpE_Abril15.pdf>\)](http://educacaosec21.org.br/wp-content/uploads/2013/08/RedeCpE_Abril15.pdf). Acesso em: 22 de novembro de 2017. Citado na página 16.
- SAS INSTITUTE INC. *Machine Learning: What it is & why it matters*. 2015. Disponível em: [\(<https://www.sas.com/it_it/insights/analytics/machine-learning.html>\)](https://www.sas.com/it_it/insights/analytics/machine-learning.html). Acesso em: 03 de Setembro de 2017. Citado 3 vezes nas páginas 4, 5 e 6.
- SIMON, P. *Too Big to Ignore: The Business Case for Big Data*. [S.l.]: Wiley, 2013. Citado na página 2.
- STANLEY, K. O. *NEAT-Python librarie*. 2015. Disponível em: [\(<http://neat-python.readthedocs.io>\)](http://neat-python.readthedocs.io). Acesso em: 22 de novembro de 2017. Citado 3 vezes nas páginas 32, 33 e 36.

STANLEY, K. O. *Neuroevolution: A different kind of deep learning - The quest to evolve neural networks through evolutionary algorithms.* 2017. Disponível em: [⟨https://www.oreilly.com/ideas/neuroevolution-a-different-kind-of-deep-learning⟩](https://www.oreilly.com/ideas/neuroevolution-a-different-kind-of-deep-learning). Acesso em: 6 de novembro de 2017. Citado na página 11.

STANLEY, K. O.; MIIKKULAINEN, R. Evolving neural networks through augmenting topologies. *Evolutionary computation*, MIT Press, v. 10, n. 2, p. 99–127, 2002. Citado 3 vezes nas páginas 19, 31 e 74.

VENNERS, B. *The Making of Python.* 2003. Disponível em: [⟨http://www.artima.com/intv/pythonP.html⟩](http://www.artima.com/intv/pythonP.html). Acesso em: 20 de novembro de 2017. Citado 2 vezes nas páginas 30 e 31.

ZHEN, J. S.; WATSON, I. D. Neuroevolution for micromanagement in the real-time strategy game starcraft: Brood war. In: SPRINGER. *Australasian Conference on Artificial Intelligence*. [S.l.], 2013. p. 259–270. Citado 2 vezes nas páginas 19 e 20.