UNIVERSIDADE DO VALE DO TAQUARI - UNIVATES

AVALIAÇÃO DA ESTRATÉGIA DE APRENDIZADO NEAT APLICADA A UM JOGO ELETRÔNICO DE CONSOLE DE 8 BITS

Alan Soder

Lajeado, julho de 2018

Alan Soder

AVALIAÇÃO DA ESTRATÉGIA DE APRENDIZADO NEAT APLICADA A UM JOGO ELETRÔNICO DE CONSOLE 8 BITS

Monografia apresentada ao Centro de Ciências Exatas e Tecnológicas da Universidade do Vale do Taquari UNIVATES, como parte da exigência para a obtenção do título de Bacharel em Engenharia da Computação.

Orientador: Prof. Dr. Marcelo de G. Malheiros

Lajeado, julho de 2018

AGRADECIMENTOS

Agradeço a minha mãe Valdina, a todo seu apoio, compreensão e educação.

Ao meu pai por toda a ajuda que me ofereceu neste longo caminho.

Ao meu irmão, que mesmo distante em alguns momentos sempre me ajudou e me fez crescer.

A minha namorada Luana por tudo que me ajudou e pelo o que passou ao meu lado sempre me apoiando e me ajudando em tudo que podia.

A minha tia Ivani e ao Paulo que confiaram em mim e foram meus fiadores.

A todos os meus amigos que sempre estiveram do meu lado para me ajudar e me apoiar.

Ao professor Marcelo, pela orientação, por ter confiado em mim e ter me auxiliado neste trabalho.

A todos os professores que me fizeram crescer, que se dedicaram a ensinar.

RESUMO

A busca pelo desenvolvimento de um programa que realize funções do cérebro humano começou junto com o início da computação eletrônica. Desde então, muitas pesquisas foram realizadas para alcançar este objetivo, em geral dentro da grande área de Inteligência Artificial. Um avanço recente foi a substancial redução do custo de equipamentos utilizados para processar redes neurais, facilitando a execução de algoritmos de aprendizado em computadores pessoais e dando novo impulso ao desenvolvimento de novos algoritmos. Um deles é o NEAT, que tem como objetivo criar uma rede neural artificial que evolui através de mudança em sua topologia e alteração de pesos das suas conexões, seguindo a abordagem de algoritmos genéticos. Esta monografia tem como objetivo avaliar o NEAT quando aplicado ao aprendizado de um jogo eletrônico. Foi escolhido para isto o jogo Gradius do Nintendo Entertainment System (NES), que funciona dentro de um emulador, interagindo e sendo controlado por uma rede neural dinâmica. Partindo de zero conhecimento, a rede neural artificial foi capaz de jogar sozinha um trecho do jogo, efetuando ações que normalmente seriam realizadas por seres humanos.

Palavras-chave: Inteligência Artificial, Aprendizado de Máquina, NEAT, jogos eletrônicos.

ABSTRACT

The search for the development of a program that performs functions of the human brain started with the beginning of electronic computing. Since then, much research has been done to achieve this goal, usually within the large area of Artificial Intelligence. A recent advance was the substantial reduction of the cost of equipment used to process neural networks, facilitating the execution of learning algorithms in personal computers and giving new impetus to the development of new algorithms. One of them is NEAT, which aims to create an artificial neural network that evolves through change in its topology and change in weights of its connections, following the approach of genetic algorithms. This work aims to evaluate NEAT when applied to the learning of an electronic game. The Gradius game of the Nintendo Entertainment System (NES) was chosen for this, which works inside an emulator, interacting and being controlled by a dynamic neural network. Starting from zero knowledge, the artificial neural network was able to play a single part of the game by doing actions that would normally be performed by humans.

Keywords: Artificial Intelligence, Machine Learning, NEAT, video games.

LISTA DE FIGURAS

Figura 1 Exemplo de Classificação Binária e Classificação multiclasse…….. 15
Figura 2 - Exemplo de Classificação por regressão..... 16
Figura 3 - Exemplo de cruzamento de um ponto de corte..... 20
Figura 4 - Exemplo de cruzamento com mais de ponto de corte..... 20
Figura 5 - Exemplo de alteração de posição de um valor..... 20
Figura 6 - Exemplos de genes e genoma..... 21
Figura 7 - Exemplo de tipos de mutação no algoritmo NEAT..... 23
Figura 8 - Exemplo de cruzamento de genoma..... 24
Figura 9 - Fórmula de cálculo de compatibilidade..... 25
Figura 10 - Fórmula de aptidão ajustada..... 26
Figura 11 - Imagem do jogo gradius onde pode ser visto a nave e seus inimigos…….. 32
Figura 12 - Esquema de comunicação entre as duas linguagens. Como em um pipe há apenas um caminho de comunicação, há necessidade de dois para haver ida e volta de informação……... 35
Figura 13 - Gráfico demonstrando evolução de tempo com a variação de gerações…….. 40
Figura 14 - Gráfico demonstrando evolução de tempo com a variação de população……. 40
Figura 15 - Gráfico demonstrando a variação de fitness com diferentes configurações de gerações…… 40
Figura 16 - Gráfico demonstrando a variação de fitness com diferentes configurações de populações……... 40
Figura 17 - Imagem demonstrando a nave parada em uma posição onde não é atingida… 41
Figura 18 - Outra imagem demonstrando a nave parada. Desta vez quase no final do cenário…….. 42

LISTA DE QUADROS

Quadro 1 - Tabela com o mapa de alguns registros da memória RAM que o jogo Gradius utiliza para gravação e leitura..... 30
Quadro 2 - Cronograma..... 33
Quadro 3 - Tabela de comandos disponibilizados e utilizados neste trabalho…… 33
Quadro 4 - Exemplo de função escrita em Lua…….. 34
Quadro 5 - Configurações utilizadas neste trabalho para a biblioteca NEAT-Python…… 37
Quadro 6 - Tabela de paços necessários para completa simulação e treinamento da rede neural…… 38
Quadro 7 - Tabela de resultados dos treinamentos da rede neural…….. 39

LISTA DE SIGLAS E ABREVIATURAS

AG	Algoritmos Genéticos
AR	Aprendizado por Reforço
AM	Aprendizado de Máquina
GPU	Graphics Processing Units
IA	Inteligência Artificial
IPC	Inter-Process Communication
NEAT	NeuroEvolution of Augmenting Topologies
NES	Nintendo Entertainment System
RNA	Rede Neural Artificial

SUMÁRIO

1 INTRODUÇÃO.....10
1.1 Objetivos.....10
1.2 Organização do trabalho.....11
2 REFERENCIAL TEÓRICO.....12
2.1 Breve histórico da Inteligência Artificial.....12
2.2 Aprendizado de Máquina.....14
2.3 Redes Neurais Artificiais.....16
2.4 Aprendizado por Reforço.....17
2.5 Algoritmo Genético.....17
2.5.1 Cromossomo.....18
2.5.2 Aptidão.....18
2.5.3 Seleção.....18
2.5.4 Recombinação.....19
2.5.5 Evolução.....21
2.6 NEAT.....21
2.6.1 Mutação.....22
2.6.2 Marca histórica.....23
2.6.3 Especiação.....25
2.6.4 Inicio mínimo.....26
2.7 Trabalhos relacionados.....27
2.7.1 Atari com aprendizado por reforço profundo.....27
2.7.2 OpenAI Dota 2.....27
2.7.3 AlphaGo Zero.....27
2.7.4 Minecraft.....28

3 DESENVOLVIMENTO.....29
3.1 Visão geral.....29
3.2 Recursos utilizados.....29
3.3 Implementação.....31
3.3.1 Emulador FCEUX.....31
3.3.2 Jogo Gradius.....31
3.3.3 Script Lua.....32
3.3.4 Comunicação.....34
3.3.5 Script Python.....35
3.3.5 NEAT- Python.....35
4 RESULTADOS.....37
4.1 Sistemática experimental.....37
4.2 Cálculo do fitness.....38
4.3 Episódios.....39
4.4 Resultados.....39
5 CONCLUSÕES.....43
5.1 Trabalhos futuros.....44
REFERÊNCIAS.....45

1 INTRODUÇÃO

Iniciada logo após a segunda guerra mundial, a Inteligência Artificial (IA) é um dos campos mais novos na área da ciência e da engenharia. Da mesma forma que a biologia molecular, IA é normalmente mencionada como “o campo que eu gostaria de estar”. Durante muito tempo tentaram entender como as pessoas pensavam e questionamentos do tipo: “como é possível que uma porção de carne pode perceber, entender e manipular coisas muito maiores e mais complicadas que si próprio” eram comuns (RUSSEL; NORVIG, 2010).

A evolução da velocidade, disponibilidade e escala reduzida das infraestruturas juntamente com a redução do custo das GPUs (Graphics Processing Units) permitiram que a IA se tornasse um dos assuntos mais populares da atualidade. Diversas técnicas que normalmente eram feitas em laboratórios especializados, agora podem ser realizadas no computador de uma pessoa comum. Esta nova realidade democratizou o acesso a um mundo antes restrito a poucos (HOOJAT, 2015).

Este trabalho pretende estudar e avaliar o algoritmo genético NeuroEvolution of Augmenting Topologies (NEAT) como técnica de aprendizado por reforço de redes neurais. Em particular, é aplicado sobre um jogo de videogame de 8 bits, funcionando em um emulador controlado por uma rede neural artificial.

1.1 Objetivos

O objetivo deste trabalho está em avaliar o algoritmo NEAT para aprendizado não-supervisionado de uma rede neural, tendo como cenário o jogo Gradius.

Os objetivos específicos deste trabalho são:

- Estudar os principais conceitos de aprendizado por reforço;
- Mapear soluções de software pré-existentes para a montagem do sistema de aprendizado;
- Aplicar o algoritmo NEAT para o jogo Gradius dentro de um sistema de emulação do NES por software;
- Avaliar os resultados obtidos.

1.2 Organização do trabalho

Este trabalho está organizado em capítulos. O primeiro capítulo representa a introdução, na qual são apresentadas uma visão geral do trabalho, com as motivações, justificativas e os objetivos a serem alcançados. No segundo capítulo, o referencial teórico é apresentado, sendo abordado assuntos pertinentes ao algoritmo NEAT e seu âmbito. Já no terceiro capítulo, explicação de como foi realizado este trabalho e o que foi utilizado no desenvolvimento. No quarto capítulo são apresentados os resultados obtidos após término do desenvolvimento. No quinto e último capítulo é apresentada a conclusão e quais são os possíveis trabalhos futuros.

2 REFERENCIAL TEÓRICO

Neste capítulo serão apresentadas as referências bibliográficas pesquisadas para o desenvolvimento deste trabalho. Serão descritos conceitos gerais e também mais específicos, contemplando as necessidades de conhecimento.

2.1 Breve histórico da Inteligência Artificial

Dentro do grande campo de Inteligência Artificial (IA), é pesquisada a possibilidade de máquinas realizarem as tarefas que humanos e animais realizam pensando, sendo uma das áreas de estudo mais recentes da ciência e engenharia. Essa ciência teve início logo após a segunda guerra mundial e sua nomeação ocorreu em 1956. Atualmente, a IA está presente em diversas áreas, indo das mais gerais às mais específicas, como no diagnóstico de doenças, na direção de automóveis, no jogo de xadrez, até na escrita de poemas. Essa ciência é relevante para qualquer área que necessite de inteligência, sendo, desta forma, um tema global (RUSSEL; NORVIG, 2010, MALLIGNTON; FUNGE, 2009).

Por milhares de anos, a humanidade tentou descobrir como pensamos, percebemos, entendemos, prevemos e manipulamos assuntos tão distintos e complicados. Em 1950, a IA começa a tomar mais forma, tendo os primeiros sistemas de computadores construídos. Na mesma época, Alan Turing começa a questionar se as máquinas poderiam pensar e, baseando-se em uma criação própria alguns anos antes (chamada "A máquina de Turing"), Turing definiu que se uma resposta de uma máquina fosse indistinguível por um ser humano, essa poderia ser considerada uma máquina inteligente (TEAHAN, 2010, RUSSEL; NORVIG, 2010, JONES, 2008; JACKSON JR, 1985).

Em meados de 1950, a IA começou a se consolidar como um campo de pesquisa. Nesse momento, muitos dos olhares estavam voltados para o que é chamado de Inteligência Artificial Sólida, sendo este conceito o de criar cópias da mente humana. Eram considerados inteligentes os sistemas que realizavam cálculos matemáticos, problemas lógicos ou ainda tinham diálogos (JONES, 2008).

Em 1957, o psicólogo Frank Rosenblatt cria um classificador linear chamado perceptron, utilizando o algoritmo de aprendizado não supervisionado. Esse algoritmo era capaz de classificar dados em duas classes separadas (RUSSEL; NORVIG, 2010, JONES, 2008).

Antes dos anos 1970, a IA já havia gerado um grande interesse e alarde pelos pesquisadores, com diversos sistemas fascinantes tendo sido desenvolvidos. Estes sistemas, porém, ficaram muito atrás das previsões feitas pelos cientistas, que acreditavam que em 1967 já iriam ter elaborado um programa que pudesse ser campeão de xadrez. Em paralelo ao desenvolvimento da IA, surge os estudos das Redes Neurais Artificiais, este sendo um estudo que visa solucionar problemas através de uma rede que aprende padrões, provendo novas formas de classificar e aprender. Entretanto, após a publicação de um trabalho chamado “Perceptron”, escrito por Marvin Minsky e Seymour Papert, que demonstrava que a técnica inventada por Rosenblatt era ruim para problemas não lineares, desta forma trazendo um declínio nas pesquisas de Redes Neurais e IA (RUSSEL; NORVIG, 2010, JONES, 2008).

A volta da IA só ocorreu no final da década de 1970, desta vez focando mais em resolver problemas específicos ao invés de tentar reproduzir a mente humana. Além disso, o retorno desta ciência trouxe também outras novas abordagens, sendo por exemplo inspiradas na biologia, como a Otimização da Colônia de Formigas (Ant Colony Optimization - ACO) de Marco Dorigo, em 1992. Em meados dos anos 80, ocorre a volta do algoritmo de *backpropagation*, sendo este aplicado em diversos problemas na ciência da computação e psicologia (RUSSEL; NORVIG, 2010, JONES, 2008).

O primeiro sistema comercial, chamado R1, só apareceu em 1980, operando na empresa Digital Equipment Corporation (DEC). Em 1986, esse sistema já havia auxiliado a empresa a poupar 40 milhões de dólares por ano. Em 1988 o grupo de IA da DEC já possuía 40 especialistas. Uma outra empresa, DuPont, já possuía 100 especialistas atuantes e estava treinando mais 500, pois apresentava uma diminuição nos custos de 10 milhões de dólares ao ano. Além disso, quase todas as maiores empresas dos Estados Unidos da América (EUA) tinham seu próprio setor de IA nessa década (RUSSEL; NORVIG, 2010).

Entre os anos de 1995 - 2001, a IA teve um crescimento exponencial, na qual um sistema inteligente ganhou do campeão mundial de xadrez, Garry Kasparov, em 1997. Além disso, em 1998 foi publicado a pesquisa de desenvolvimento da primeira rede neural, utilizada para reconhecimento de escrita a mão. Em 2007, Fei Fei Li e seus colegas começaram a construir a plataforma de pesquisa em imagens chamada ImageNet, que atualmente é muito utilizada para treinamento de redes neurais profundas para o reconhecimento de objetos, pessoas e animais (DENG et al, 2009; LECUN et al, 1998; PRESS, 2016).

2.2 Aprendizado de Máquina

Aprendizado de Máquina (AM) é um método que visa automatizar a criação de modelos analíticos. É uma das linhas da Inteligência Artificial que se apoia na ideia de sistemas de aprendizado automáticos através de dados, identificação de padrões e tomadas de decisões sem ou quase sem interferência humana (SAS, 2018).

Nas últimas três décadas, Aprendizado de Máquina se tornou uma das principais áreas de tecnologia, muitas vezes sem ser percebida. Um dos motivos disto é o grande aumento das bases de dados categorizados que estão disponíveis.

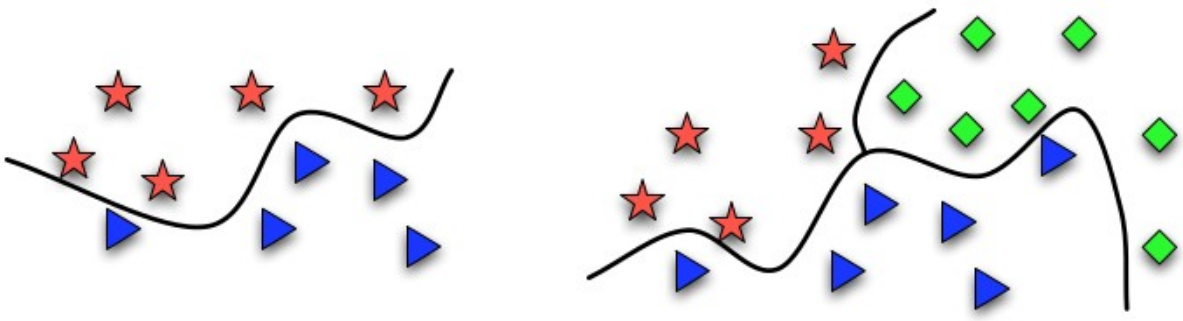
Aprendizado de Máquina pode ser utilizado para diversos fins, e dentre eles os mais conhecidos são os de motores de buscas, como o utilizado pelo Google para aprimorar pesquisas; filtragem colaborativa (*collaborative filtering*) da Amazon e Netflix que realizam indicações de produtos e filmes, além da tradução automática, técnica utilizada pelo parlamento do Canadá (SMOLA; VISHWANATHAN, 2008).

É vasta a dimensão de problemas que podem ser resolvidos com AM. Pesquisadores identificaram um crescimento de modelos que podem ser utilizados para resolução de um grande conjunto de situações. Dentre os problemas mais comuns enfrentados com AM, estão:

- **Classificação binária:** Foi a principal forma de pesquisas e desenvolvimentos teóricos no último século, sendo o problema mais estudado em AM. De uma forma mais simples, classificação binária reduz a pergunta : "Dado um padrão x desenhado a partir de um domínio X, estime o valor que variável aleatória assumirá". Por exemplo, dado uma maçã e uma laranja, é esperado que o programa diga se é uma maçã ou uma laranja. Exemplo na figura 1 (SMOLA; VISHWANATHAN, 2008).

- **Classificação multiclasse:** É uma extensão lógica de classificação binária. A principal diferença, é que este método possui diversas possibilidades de classificação que vão além de apenas entender se o que é apresentado é *x* ou *y*, pois esperam uma classificação de *n* possibilidades. Um exemplo desta classificação pode ser um documento que esperamos saber a que linguagem pertence: português, inglês, espanhol, alemão, etc. Exemplo na figura 1. (SMOLA; VISHWANATHAN, 2008).

Figura 1 - Exemplo de Classificação Binária e Classificação Multiclasse.

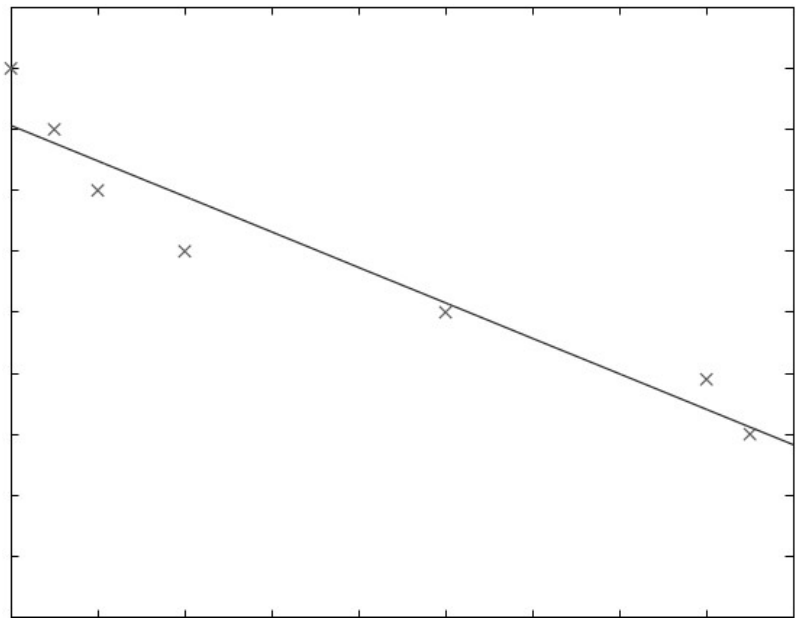


Fonte: (SMOLA; VISHWANATHAN, 2008).

- **Regressão:** O objetivo desta é estimar um valor real para uma variável *y* dado um modelo *x*. Um exemplo que pode ser visto neste método é o de previsão de valores de

uma ação (bolsa de valores). A principal diferença entre os problemas que utilizam regressão é o cálculo da perda. Exemplo na figura 2 (SMOLA; VISHWANATHAN, 2008).

Figura 2: Exemplo de uma classificação por regressão.



Fonte: (ALPAYDIN, 2010).

2.3 Redes Neurais Artificiais

Uma Rede Neural Artificial (RNA) é um sistema paralelo e distribuído, composto de unidades de processamento triviais (neurônios artificiais) alocados em uma ou diversas camadas interconectadas por um grande número de conexões com o objetivo de calcular funções matemáticas. Estas ligações normalmente estão integradas a pesos, esses utilizados para armazenar o conhecimento e avaliar todas as entradas de cada neurônio (BRAGA et al, 2011).

As RNAs são atrativas devido ao seu poder de aprendizagem por meio de exemplos e da generalização da informação obtida. A generalização está ligada à aptidão da rede em aprender por intermédio de um conjunto de amostras e, em seguida, apresentar respostas para informações não apresentadas anteriormente. Além disso, outra propriedade que faz das

RNAs importantes é a eficácia de se auto-organizar e a possibilidade de processamento temporal (BRAGA et al., 2011).

2.4 Aprendizado por Reforço

Aprendizado por Reforço (AR) é, simultaneamente, um problema, a solução para resolução de problemas e a área de estudo que pesquisa deste assunto. Problemas de AR envolvem descobrir como estruturar ocorrências em ações e, também, como maximizar a indicação de recompensa. Essencialmente, esses problemas necessitam ser repetitivos, pois o sistema de aprendizado é influenciado pelos resultados das ocorrências anteriores (SUTTON; BARTO, 2016).

Diferente das formas normais de Aprendizado de Máquina, AR não especifica quais ações adotar e sim as que deram melhores resultados após a tentativa. Além disto, todas as ações tomadas por uma rede de AR refletem em futuros eventos (SUTTON; BARTO, 2016).

Um dos desafios de AR é o exploração de ações anteriores. Para conseguir alcançar uma indicação de recompensa alta, AR deve preferir ações já tomadas no passado e ser efetivo para conseguir recompensa. Para isto, é necessário que sejam selecionadas ações não escolhidas anteriormente. É necessário aproveitar-se de oque já foi utilizado na tentativa de conseguir recompensa e , ao mesmo tempo, é imprescindível explorar novas possibilidades na tentativa de conseguir melhores seleções no futuro. A questão mais importante aqui é que nem exploração nem tirar proveito podem ser únicos fatores para obtenção de recompensa. É necessário variar entre essas duas formas e observar quais delas obtém o melhor resultado (SUTTON; BARTO, 2016).

2.5 Algoritmo Genético

A ideia de Algoritmo Genético (AG) teve sua primeira aparição em 1967, na tese de J. D. Bagley, intitulada “The Behavior of Adaptive System Which Employ Genetic and

Correlative Algorithms", sendo fortemente influenciada por J. H. Holland, o pioneiro dos algoritmos genéticos (KRAMER, 2017).

Um AG tem como o objetivo a otimização e a busca da solução ótima para uma vasta série de problemas. O conceito de AG é baseado em evolução e isto é devido ao grande sucesso e variedade de espécies encontradas no mundo. A existência destas espécies ocorreu devido a capacidade destas de se adequarem ao seu ambiente (KRAMER, 2017).

Os AGs são estruturados em um número específico de itens, baseados na genética e na evolução natural. Isto é um dos pontos fortes dos AGs, pois significa que componentes comuns podem ser reutilizados em diversos problemas diferentes. Os principais componentes são: codificação do cromossomo, a função de aptidão, seleção, recombinação e o esquema de evolução (MCCALL, 2005).

2.5.1 Cromossomo

Algoritmos Genéticos manipulam populações de cromossomos que são representações em formato de texto de soluções para problemas. Um cromossomo é uma abstração de um DNA biológico, que pode ser representado por um conjunto de letras do alfabeto. Uma posição em um cromossomo é chamado de gene e a letra nesta posição é chamada de alelo (MCCALL, 2005).

2.5.2 Aptidão

A função de aptidão (em inglês, *fitness*) é utilizada para medir a qualidade de um cromossomo como solução para um problema específico. Em comparação com biologia, um cromossomo é descrito como um genótipo, enquanto que a solução do problema como fenótipo (MCCALL, 2005).

2.5.3 Seleção

Enquanto que Algoritmos Genéticos utilizam a aptidão como um medidor de qualidade apresentado pelos cromossomos em uma população, o elemento de seleção é eleito para usar aptidão para guiar a evolução de um cromossomo. Desta forma, cromossomos são selecionados para recombinação baseados em sua aptidão e aqueles com maior aptidão tem maior chance de serem escolhidos, fazendo com que os mais aptos sejam selecionados (MCCALL, 2005).

A forma mais comum de seleção é a baseada em sua aptidão. Além desta, existem outras formas de seleção, como: estocástica aleatória, por torneio e por truncamento (MCCALL, 2005).

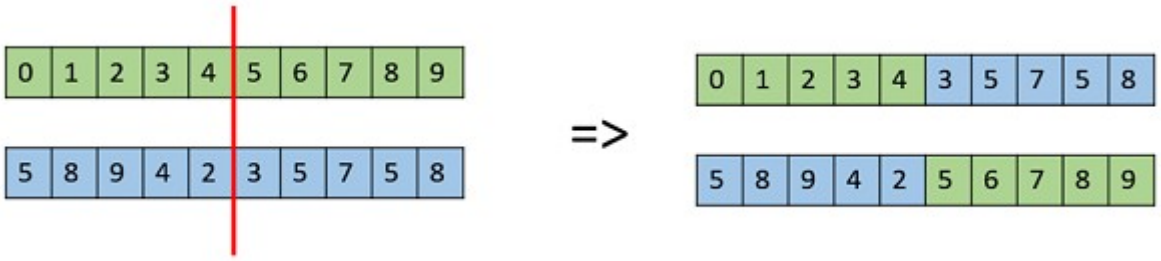
2.5.4 Recombinação

Recombinação é o processo no qual cromossomos são selecionados em uma população e combinados para formar as próximas populações. Neste processo, há a garantia de que os cromossomos mais aptos sejam evoluídos devido a seleção ter a maior probabilidade de buscar os melhores (MCCALL, 2005).

Na recombinação há dois processos principais: o cruzamento e a mutação. Estes processos não são exatos e cada um tem uma chance probabilística de acontecer. O cruzamento de cromossomos representa a mistura de valores de dois cromossomos-pais para produzir um ou dois cromossomos-filhos. Após dois serem escolhidos, um número aleatório com chance igual entre 0 e 1 é gerado e comparado com uma taxa de cruzamento previamente informada. Caso este número aleatório seja maior que a taxa de cruzamento, nenhum cruzamento é realizado e um ou os dois cromossomos-pais continuam inalterados e seguem para para a próxima recombinação. Caso o número aleatório seja menor que a taxa de cruzamento, esta operação é realizada (MCCALL, 2005).

Uma das formas de cruzamento é a de apenas um ponto, que pode ser observada na figura 3. Um ponto de cruzamento entre 0 e n (n sendo o total de genes) é escolhido com chances iguais e, a partir deste ponto, dois cromossomos-filhos são gerados. As informações de ambos os cromossomos são invertidos após o ponto de cruzamento (MCCALL, 2005).

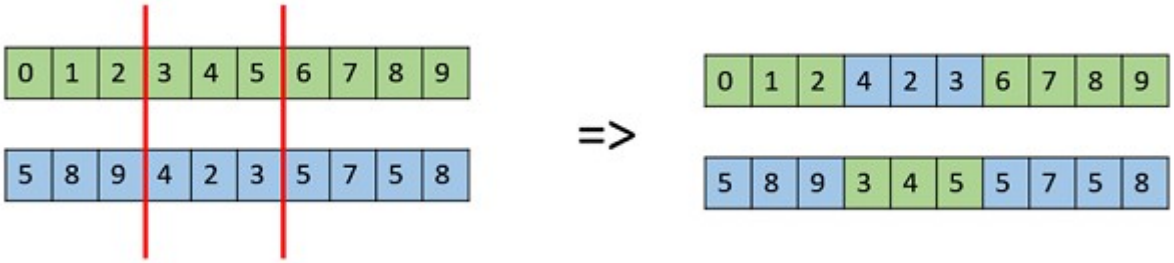
Figura 3: Representação de cruzamento de um ponto.



Fonte: (TUTORIALSPOINT, 2017).

Além desta, existem outras formas de cruzamento, como a seleção de dois ou mais pontos, como visto a seguir na figura 4:

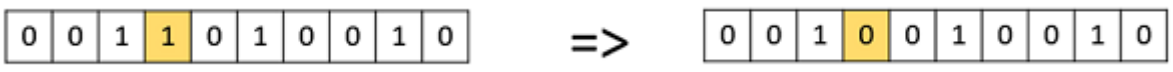
Figura 4: Exemplo de cruzamento selecionando mais de um ponto.



Fonte: (TUTORIALSPOINT, 2017).

A operação de mutação, por sua vez, age diretamente em um cromossomo para alterar os valores de um ou mais alelos, como por exemplo na figura 5. Para cada posição de um cromossomo, um número aleatório é gerado e comparado com uma taxa de mutação pré selecionada. Não há mutações se o número aleatório gerado for maior que a taxa de mutação. Já se o número for menor ou igual a taxa de mutação, então o valor daquela posição é alterado. As taxas de mutação geralmente são bem baixas, como por exemplo 0,001 (MCCALL, 2005).

Figura 5: Exemplo de alteração de valor de uma posição.



Fonte: (TUTORIALSPOINT, 2017).

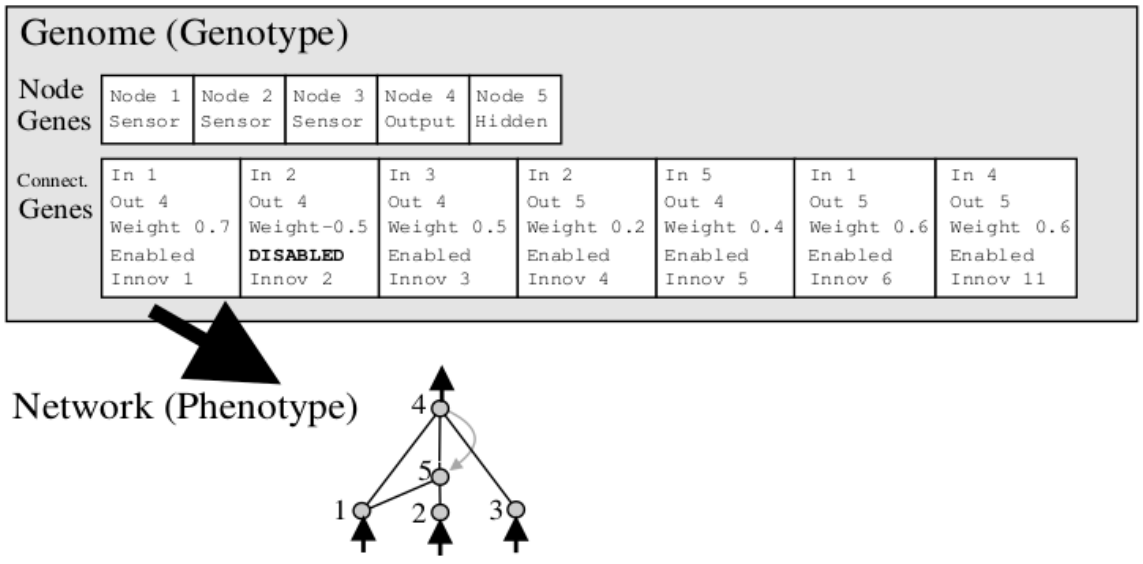
2.5.5 Evolução

Após a recombinação, os cromossomos resultantes passam a fazer parte da população posterior, e o processo de seleção e recombinação são repetidos até que se complete uma nova geração de população. Desta forma, o AG é continuado até que a melhor aptidão (objetivo) seja alcançado ou até que um máximo de gerações (pré-especificadas) seja gerada (MCCALL, 2005).

2.6 NEAT

NeuroEvolution of Augmenting Topologies (NEAT) é um algoritmo genético que visa tirar vantagem de uma rede por meio de sua estrutura, visando diminuir o tamanho da pesquisa dos pesos das conexões de uma rede neural. Caso a estrutura evolua para que a topologia da rede seja minimizada, e cresça ao longo do tempo, ganhos significativos em velocidade são apresentados (STANLEY; MIIKKULAINEN, 2002). Na figura 6 é possível ver a estrutura da rede neural para o algoritmo NEAT.

Figura 6: Imagem demonstrando um exemplo das informações dos genes e do genoma.



Fonte: (STANLEY; MIIKKULAINEN, 2002).

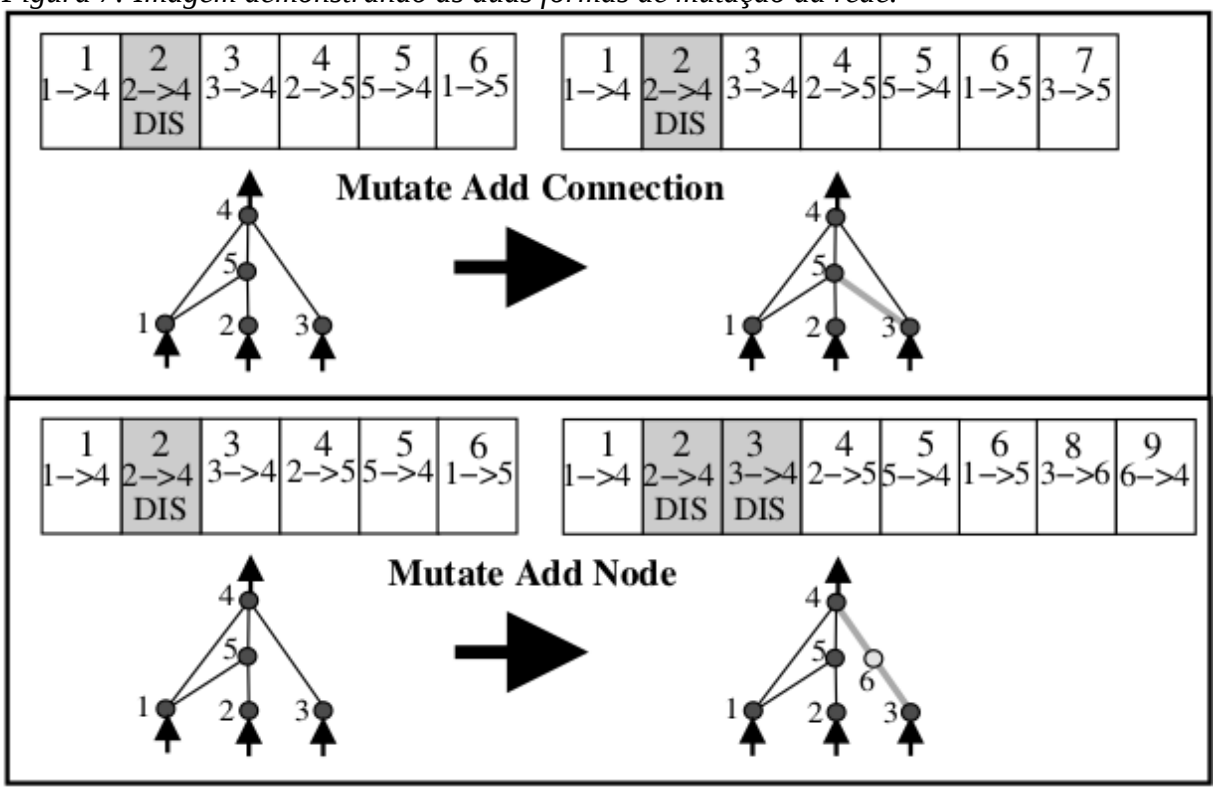
NEAT é designado para que os genes possam ser colocados em ordem quando há cruzamento. Genomas são representações de conectividade da rede. Cada genoma possui uma lista de genes de conexão, cada um referenciando a dois genes de nó conectados. Genes de nó fornecem uma lista de entradas, contendo nós escondidos, e de saídas que podem ser conectadas. Cada gene de conexão possui um nó de entrada, um nó de saída, seus respectivos pesos, a habilidade de estar ou não ativado e um número de inovação, que permite encontrar os genes correspondentes (STANLEY; MIIKKULAINEN, 2002).

2.6.1 Mutação

Em NEAT, a mutação pode ocorrer de duas formas (demonstradas na figura 7): alterando pesos das conexões ou da estrutura da rede. A mutação dos pesos ocorre da mesma forma que ocorre em algoritmos genéticos (cruzamento e recombinação). Já a mutação da rede ocorre de duas formas, na qual ambas incrementam os genes: adicionando uma conexão ou um novo nó. Na mutação por meio de adição de conexão, um novo gene de conexão com peso aleatório é incluído, conectando dois nós anteriormente desconectados. Na mutação por adição de nó, uma conexão existente é dividida e um novo nó é colocado onde a antiga conexão estava. A conexão antiga então é desabilitada e duas novas conexões são incorporadas ao genoma. A nova conexão ligada ao novo nó recebe peso de 1 e a conexão ligada ao antigo nó recebe o mesmo peso da antiga ligação (STANLEY; MIIKKULAINEN, 2002).

Os genomas irão gradativamente aumentar por meio destas mutações. Genomas de tamanhos diferentes serão produzidos, às vezes, com conexões diferentes nas mesmas posições (STANLEY; MIIKKULAINEN, 2002).

Figura 7: Imagem demonstrando as duas formas de mutação da rede.



Fonte: (STANLEY; MIIKKULAINEN, 2002).

2.6.2 Marca histórica

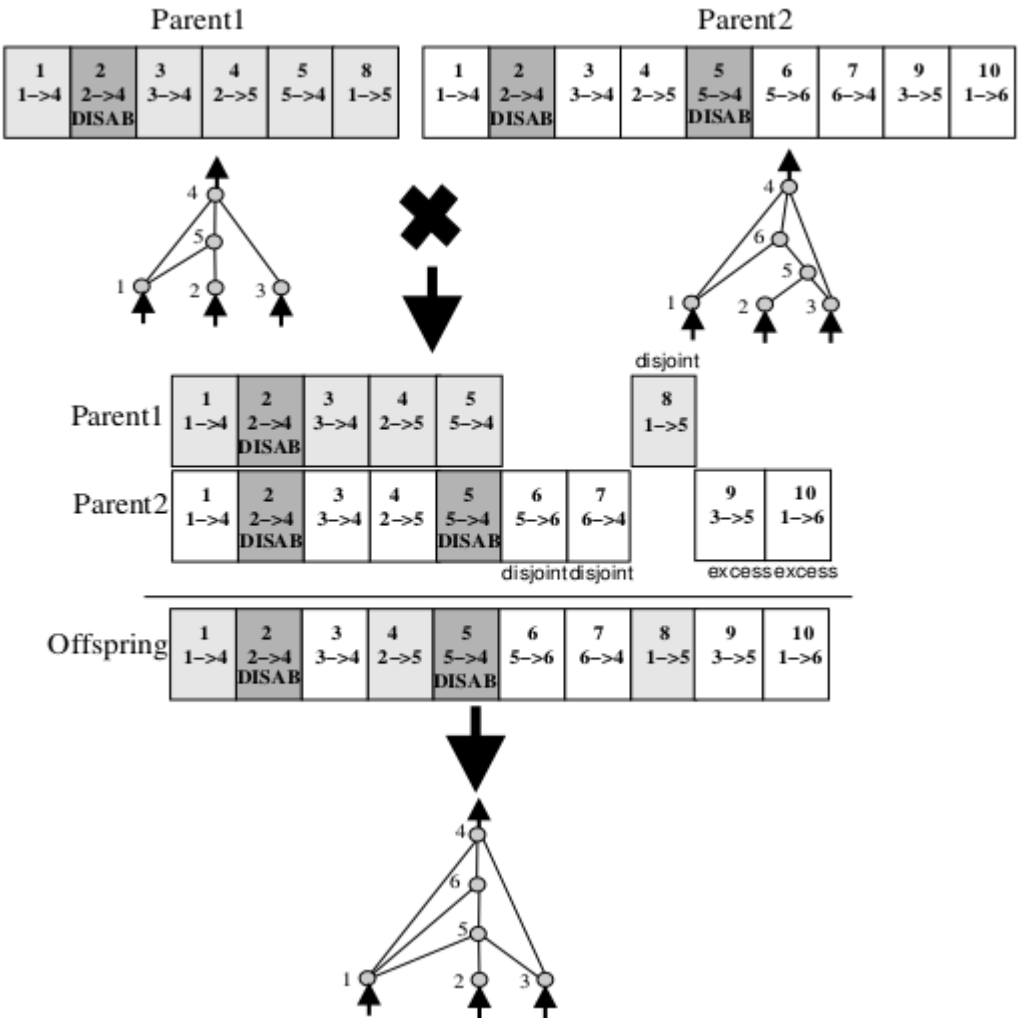
Para poder realizar cruzamento de redes de tamanhos diferentes, NEAT necessita seguir a evolução por intermédio de um número histórico. Este número é chamado de *innovation number* (número de inovação). Este é incrementado toda vez que um novo gene é criado, por meio da mutação da estrutura da rede, sendo então, atribuído a este gene. Desta forma, estes números formam a ordem cronológica da criação destes genes (STANLEY; MIIKKULAINEN, 2002).

Esta informação de inovação da ao NEAT uma capacidade poderosa. O sistema conhece exatamente quais genes se combinam. Quando é realizado o cruzamento, os genes com mesmo número de inovação de ambos os genomas são alinhados. Estes, por sua vez, são chamados de genes correspondentes. Genes que não são correspondentes são chamados de deslocados ou excesso, dependendo de onde aparecem (dentro ou fora do alcance dos seus cromossomos-pais) (STANLEY; MIIKKULAINEN, 2002).

No processo de cruzamento, genes específicos são escolhidos aleatoriamente dentre os genes que combinam (mesmo número de inovação), e nos genes que não combinam, são escolhidos os genes do genoma dos cromossomos-pai que forem mais aptos (STANLEY; MIIKKULAINEN, 2002).

Estas informações podem ser vistas na seguinte figura:

Figura 8: Exemplo de cruzamento de dois genomas.



Fonte: (STANLEY; MIIKKULAINEN, 2002).

2.6.3 Especiação

Para realizar a proteção de novas estruturas topológicas, estas são especiadas, permitindo que possam evoluir competindo apenas com seu nicho. A ideia é separar a população em espécies com topologias similares estejam na mesma espécie (STANLEY; MIIKKULAINEN, 2002).

O número de genes deslocados e excessivos entre um par de genomas é a medida natural de suas distâncias de compatibilidade. Quanto mais deslocados dois genomas são, menor o histórico de evolução eles dividem e, portanto, menor sua compatibilidade. Desta forma, a distância de compatibilidade δ de diferentes estruturas pode ser calculada com combinações lineares do número de genes excessivos E e deslocados D, além do peso médio dos genes correspondentes (incluindo os genes desabilitados), como mostrado na fórmula da figura 9 (STANLEY; MIIKKULAINEN, 2002).

Figura 9: Fórmula do cálculo de compatibilidade.

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot \overline{W}.$$

Fonte: (STANLEY; MIIKKULAINEN, 2002).

A medida da distância permite que possa ser realizada a especiação utilizando um limite de compatibilidade δ. Uma lista ordenada de espécies é mantida e, em cada geração, genomas são colocados sequencialmente em espécies. Cada espécie existente é formada por um número aleatório dentro das espécies da geração anterior. Um determinado genoma *g* na geração atual é colocado na primeira espécie que *g* é compatível com o genoma representativo daquela espécie, não permitindo sobreposição das espécies. Caso *g* não se encaixe com nenhuma espécie, uma nova espécie é criada com *g* sendo seu representativo (STANLEY; MIIKKULAINEN, 2002).

Como meio de reprodução, NEAT utiliza *explicit fitness sharing*, fórmula criada por Goldberg e Richardson em 1987 (figura 10). Está fórmula faz com que organismos da mesma espécie precisem compartilhar a aptidão de seu nicho. Desta forma, uma espécie não consegue ficar muito grande, mesmo que seu organismo seja bom. Assim sendo, nenhuma espécie assumirá o controle de toda a população. A aptidão ajustada $f_i^{'}$ para o organismo *i* é calculado de acordo com a sua distância δ comparada com todos os outros organismos *j* na população, de acordo com a figura 10 (STANLEY; MIIKKULAINEN, 2002):

Figura 10: Fórmula do explicit fitness sharing.

$$f_i^i = \frac{f_i}{\sum_{j=1}^{n} sh(\delta(i,j))}$$

Fonte: (STANLEY; MIIKKULAINEN, 2002).

Portanto, $\sum_{j=1}^{n} sh(\delta(i,j))$ minimiza a quantidade de organismos na mesma espécie do organismo *i*. Esta redução é natural desde que as espécies já estejam agrupadas por compatibilidade utilizando o limitador δ_t. Um potencial número diferente de descendentes é designado para cada espécie, sendo proporcional a soma da aptidão ajustada $f_i^{'}$ de seus organismos membros. Espécies então reproduzem primeiramente eliminando os membros de menor performance da população. Depois disto, toda a população é substituída pelos descendentes dos organismos restantes em cada espécie (STANLEY; MIIKKULAINEN, 2002).

2.6.4 Inicio mínimo

NEAT inicia seu processo com espaços dimensionais mínimos, começando com uma população de redes uniforme, que não possuem nós ocultos (todas entradas conectadas às saídas). Novas estruturas são introduzidas por meio de mutações e apenas continuam aquelas que tiveram a aptidão alta. Esta diminuição do tamanho inicial da rede oferece uma velocidade aumentada quando comparada com outras abordagens (STANLEY; MIIKKULAINEN, 2002).

2.7 Trabalhos relacionados

Nesta seção serão apresentados trabalhos relacionados. Estes não necessariamente utilizam apenas o algoritmo NEAT mas utilizam redes neurais com treinamento sendo realizados por outras técnicas, uma vez que a aplicação de técnicas de aprendizado em jogos a jogos de videogame é recente e não há tantos materiais de referência.

2.7.1 Atari com aprendizado por reforço profundo

Pesquisadores do Deepmind Technologies treinaram uma rede neural para aprender a jogar jogos de Atari 2600. Em seu desenvolvimento, utilizaram Aprendizado por Reforço Profundo e uma variação do algoritmo Q-learning. Os experimentos foram realizados em sete jogos, sendo eles: Beamrider, Breakout, Enduro, Pong, Q*bert, Seaquest e Space Invaders. A técnica utiliza com entrada da rede neural as posições da tela (84 por 84 *pixels*) e as saídas são as ações a serem tomadas pelo jogo. Os resultados do trabalho foram que dentre os sete jogos testados, três obtiveram melhores resultados que os dos humanos (MNIH et al, 2013).

2.7.2 OpenAI Dota 2

Uma das mais famosas e mais recentes conquistas obtidas com o uso de redes neurais foi o desenvolvimento de um sistema que conseguiu jogar Dota 2 e derrotar o jogador considerado o melhor do mundo neste jogo. Em uma seção de melhor de três, o *bot* criado pelos pesquisadores da OpenAI conseguiu ganhar duas vezes seguidas. Em entrevista, um dos desenvolvedores afirmou que o programa derrotou diversos outros jogadores, onde o *bot* não perdia nenhum jogo (OPENAI, 2017).

2.7.3 AlphaGo Zero

AlphaGo Zero foi o primeiro sistema a conseguir consistentemente derrotar humanos no jogo Go. Ele consiste em uma evolução do AlphaGo comum que aprendia com redes neurais e aprendizado supervisionado. Esta nova versão foi criada para aprender através das próprias jogadas e, então, conseguir treinar o suficiente para derrotar humanos (SILVER et al, 2017).

2.7.4 Minecraft

Foram aplicados treinamentos de rede neural e algoritmos de aprendizado de máquina para fazer com que um personagem do jogo Minecraft podesse se movimentar e conseguisse realizar ações normalmente feitas por humanos. Para concretizar isso, os pesquisadores criaram um algoritmo chamado *Hierarchical Deep Reinforcement Learning Network (H-DRLN)*, que possibilitou o aprendizado de novas técnicas a partir de aprendizados anteriores, como por exemplo a utilização do aprendizado de navegação para o de posicionamento de blocos (TESSLER et al, 2017).

3 DESENVOLVIMENTO

Neste capítulo são apresentados os recursos e princípios para a realização do presente trabalho. Serão apresentadas informações gerais e específicas do que foi utilizado e do objetivo a ser alcançado.

3.1 Visão geral

O objetivo deste trabalho é o desenvolvimento um sistema para avaliação do algoritmo NEAT aplicado no jogo eletrônico Gradius do Nintendo Entertainment System. Isso é feito através de *scripts* escritos nas linguagens Lua e Python, utilizando emulação através do emulador FCEUX.

3.2 Recursos utilizados

Para a realização deste trabalho, foram utilizadas ferramentas disponíveis gratuitamente, mas que apresentam ótimas funcionalidades, sendo estas essenciais para o este desenvolvimento. As ferramentas e componentes utilizados foram:

- Computador pessoal: o computador utilizado para o desenvolvimento foi um *desktop* contendo um processador Core I7-4790 com 4,0 GHz com quatro núcleos de processamento, 8 GB de memória RAM, um SSD Kingston de 120 GB com velocidades de leitura de 550 MB/s e escrita 500 MB/s. O sistema operacional utilizado foi o Ubuntu versão 16.04.

- Controle *joystick* e conversor: para realizar os testes foi utilizados um controle Fr-201, que é um controle genérico do console PlayStation 2 da Sony. Além deste, para a conexão através de USB foi utilizado um adaptador KP 3454 Knup.

Além destes componentes, foi utilizada uma imagem ROM do jogo Gradius para ser executada pelo FCEUX. É apresentado no quadro 1 algumas posições de memória utilizadas neste trabalho, préviamente mapeadas pelo site *Datacrystal :*

Quadro 1: Tabela com o mapa de alguns registros do memória RAM que o jogo Gradius utiliza para gravação.

Posição de memória RAM	Função	Detalhes
0x0007	Botão pressionado	Botão pressionado no momento
0x000F	Posição do cursor na tela inicial	
0x0015	Informação de jogo parado	0 = não e 1 = sim
0x0020	Quantidade de vidas do jogo atual	Começa com 3 vidas e quando a última é utilizada, é colocado 255 para informação que terminou.
0x0040	Velocidade da nave	
0x0041	Mísseis disponíveis	
0x0042	Posição da barra de poder	Vai de 0-6, ao passar de 6, volta para 0
0x0044	Tipo de arma	0 = Normal, 1 = Laser e 2 = Tiro dobrado
0x0045	Número de opções	Deve-se manter em 02.
0x0046	Força da proteção	Ao obter escudo, fica com proteção de 5 contatos
0x004C	Tempo antes da transição da tela	
0x0060	Carregamento dos inimigos	2 = inimigos carregando e atacando e 0 = personagem morto
0x0100	Determina se o jogador está vivo	1 = vivo e 2 = morto
0x07A0 - 0x07B7	Posição horizontal (x) da nave	Da mais antiga para a mais nova posição
0x07B0 - 0x07C7	Posição vertical (y) da nave	Da mais antiga para a mais nova posição
0x07E4	Menor dígito da pontuação	A cada ponto, a pontuação aumenta em 100
0x07E5	Dígito do meio da pontuação	A cada ponto, a pontuação aumenta em 1000
0x07E6	Maior dígito da pontuação	A cada ponto, a pontuação aumenta

Fonte: (DATACRYSTAL, 2017).

3.3 Implementação

O desenvolvimento deste trabalho foi realizado com as linguagem Python e Lua, realizando comunicação entre essas linguagens através da técnica de *inter-process communication* (IPC) utilizando *named pipes*. Para executar o jogo foi utilizado o emulador FCEUX. Nas próximas seções serão explicados mais detalhadamente cada tecnologia.

3.3.1 Emulador FCEUX

Para poder executar o jogo Gradius no computador foi utilizado o emulador FCEUX versão 2.2.2. FCEUX é um emulador capaz de reproduzir os consoles NES, Famicom Disk System (FDS) e Dendy. Este emulador suporta os formatos de vídeo NTSC (EUA/Japão), PAL (Europa) e NTSC-PAL híbrido. Além disso, reconhece *scripts* em linguagem Lua. O interpretador Lua permite realizar operações com as memórias RAM e VRAM, como também realizar de forma programada ações que normalmente seriam realizadas pelo controle conectado ao console.

3.3.2 Jogo Gradius

Gradius é um jogo do estilo s*hoot'em up* de deslocamento horizontal e foi desenvolvido pela Konami em 1985 para o NES. Neste jogo o jogador pilota uma nave conhecida como Vic Viper que deve se defender de naves alienígenas.

Neste jogo pode-se movimentar, disparar projéteis, pegar e utilizar os bônus que são deixados pelos inimigos quando morrem. Para facilitar o desenvolvimento e aprendizado da rede neural, os bônus não foram utilizados neste trabalho. Na figura 11 é demonstrado um exemplo do jogo.

Figura 11: Imagem do jogo gradius onde pode ser visto a nave e seus inimigos.

Fonte: Autor (2018).

3.3.3 Script Lua

Todas as interações entre o emulador e o jogo Gradius foram realizadas através de um *script* Lua, desenvolvido para utilizar as funções disponibilizadas pelo emulador. Através deste *script* é possível obter as informações necessárias para treinamento da rede com base no conteúdo das memórias RAM e VRAM do console, realizar comandos para movimentar a nave e ainda obter informações do que é mostrado na tela, no caso deste trabalho, informações sobre os inimigos.

Para a realização deste trabalho, as seguintes informações (Quadro 2) foram obtidas da memória do emulador:

Quadro 2: Informações extraídas com script Lua.

Informação	**Posição da memória/função**
frame atual	emu.framecount()
posição X VIC	0x07B7
posição Y VIC	0x07C7
se a nave está viva ou morta	0x0100
número de identificação inimigos	0x0307 - 0x0320
posições X dos inimigos	0x0367 - 0x0380
posições Y dos inimigos	0x0327 - 0x0340

Fonte: Autor (2018)

Todas essas informações são obtidas através do *script* Lua funcionando dentro do próprio emulador. Para a realização deste trabalho, os seguintes comandos do emulador foram utilizados:

Quadro 3: Tabela de comandos disponibilizados e utilizados neste trabalho.

Comando	**Descrição**
emu.frameadvance()	Avança um *frame* do jogo. É utilizado em toda rodada de envio de *inputs* para rede neural e retorno dos *outputs* da rede neural.
joypad.set(int player, table input)	Designa quais serão utilizados pelo jogador selecionado. Neste trabalho, as saídas do rede neural foram passadas para esta função. Desta forma pode-se mover a nave e atirar. Também foi utilizado para passar os menus do jogo, colocando como comando o botão *start.* Exemplo de entrada: {0,0,0,0,0,0,1,0} onde 1 significa pressionado e 0 não pressionado.
memory.readbyte(int address)	Comando utilizado para leitura dos dados da memória RAM e VRAM. Todos as entradas da rede neural foram obtidas através deste comando, selecionando a posição do dado necessário.
emu.speedmode(string mode)	Com este comando foi possível selecionar o modo *maximum* para ter uma velocidade acima da normal (3200%). Isso foi essencial no treinamento da rede, pois foi possível treinar a rede muito mais rápido que normalmente seria.

Fonte: Autor (2018)

Em Lua também há comandos para leitura e escrita de arquivos, necessários para comunicação entre os *scripts* Lua e Python. Os comandos para leitura e escrita dos *pipes* foram **io.open(path, mode)** para abrir o arquivo e **arquivo:read()** para leitura. Para escrita, os comandos utilizados foram **arquivo:write(string data)**, **arquivo:flush()** e **arquivo:close()**.

Um exemplo de código para escrita de arquivo (neste caso o *pipe*) é o seguinte:

Quadro 4: Exemplo de função escrita em Lua.

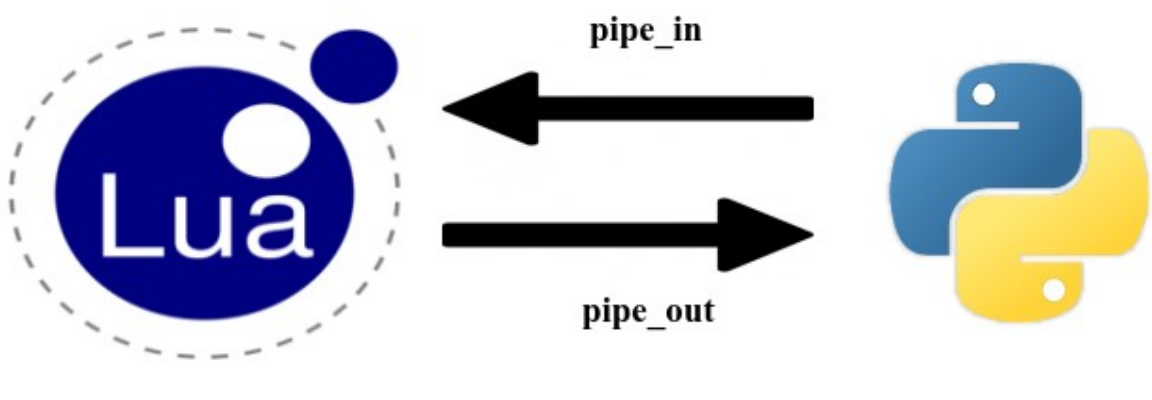
```
function write_to_pipe(data)
   pipe_out, _, _ = io.open(pipe_in_path, "w");
   if data and pipe_out then
      pipe_out:write(data);
      pipe_out:flush();
      pipe_out:close();
   end;
   return;
end;
```

Fonte: Autor (2018).

3.3.4 Comunicação

A comunicação entre o *script* de Python e o *script* de Lua é realizada através de IPC, utilizando *named pipes*. *Named pipes* são dispositivos de comunicação unidirecional que permitem escrita de bytes em um processo e a leitura em outro(s) (RITCHIE; THOMPSON, 1974). Um exemplo do esquema de comunicação utilizado no trabalho pode ser visto na figura 12 a seguir:

Figura 12: Esquema de comunicação entre as duas linguagens. Como em um pipe há apenas um caminho de comunicação, há necessidade de dois para haver ida e volta de informação.



Fonte: Autor (2018).

3.3.5 Script Python

O desenvolvimento em Python visou a criação da inteligência deste trabalho. Foi desenvolvida a integração com o módulo NEAT-Python para treinamento da rede neural com os dados recebidos através do *pipe* processados com o *script* Lua.

Neste trabalho, Python foi essencial pois possui diversos módulos auxiliares prontos para utilização, como as bibliotecas **subprocess** e **distutils** que foram essenciais pois disponibilizam o acesso à abertura e controle de outros programas, como por exemplo o emulador FCEUX.

Para controle da comunicação foi utilizado o módulo **os** que possui a função **mkfifo**, utilizada para criação de um *pipe*. Para leitura e escrita dos *pipes*, foram utilizadas funções próprias do Python de manipulação de arquivos, como **open** para abrir, **write** para escrita e **read** para leitura.

3.3.5 NEAT- Python

NEAT-Python é uma biblioteca escrita em Python para facilitar a utilização do algoritmo NEAT. Esta biblioteca foi utilizada neste trabalho para realizar o treinamento da rede neural e a avaliar o seu desempenho. Para efetuar um treinamento mínimo, é necessário

criar um arquivo de configuração da rede, onde são preenchidos, entre outras coisas, o número de entradas da rede, a quantidade de saídas, a população e função de cálculo de ativação. Além disso é necessário construir uma função para treinamento da rede e cálculo de *fitness*.

Esta bibliotéca também possui funções de controle de crescimento da rede neural, aplicando o conseito de *stagnation*. Através de parâmetros na configuração é possível definir quanto tempo um "espécie" ficam na rede, sendo removidas se ficarem mais gerações que o definido nestes parâmetros.

4 RESULTADOS

Neste capítulo será apresentado como as simulações foram realizadas, quais parâmetros foram utilizados para treinamento da rede neural, como foi realizado o cálculo do *fitness* e quais foram os resultados obtidos com este treinamento.

4.1 Sistemática experimental

Para realizar um treinamento de uma rede neural utilizando o pacote NEAT-Python foram definidos os parâmetros mostrados no Quadro 5, configurados no arquivo de configuração da rede:

Quadro 5: Configurações utilizadas neste trabalho para a biblioteca NEAT-Python.

Campo da configuração	**Valor**	**Descrição**
fitness_criterion	Max	Critério para obtenção do *fitness*
fitness_threshold	10000	Limite do *fitness*. Ao alcançar o sistema chama a função **found_solution.**
pop_size	5, 20, 50, 100	Foram utilizados quatro valores de população nos testes.
num_inputs	43	Número de entradas da rede neural. Foram utilizadas os ID, posições *x* e *y* da nave e dos inimigos e um cálculo de distância. Além disso, o *frame* atual também é enviado para a rede.
num_outputs	5	Número de saídas da rede neural. As saídas foram os quatro botões de movimentação (esquerda, direita, cima, baixo) e disparo.

Fonte: Autor (2018).

Esses parâmetros foram definidos depois de alguns testes de treinamento para ver quais eram necessários. O parâmetro do tamanho da população foi definido desta forma para haver uma comparação na evolução deste número.

Para realizar um treinamento completo, são necessários diversos passos e esses são:

Quadro 6: Tabela de passos necessários para completa simulação e treinamento da rede neural.

Passos	**Explicação**
Abrir o emulador FCEUX através do módulo *subprosses* na linguagem Python.	Este é o primeiro passo e necessário para que a simulação comece. Ele abre o emulador FCEUX e designa as configurações para abrir também o código Lua.
Realizar a criação dos *pipes* e então começar a comunicação e simulação do jogo.	Através dos *pipes* é que o jogo começa a simular, realizando os primeiros comandos, passando o meno do jogo.
Realização do treinamento da rede neural.	Agora que o jogo está sendo simulado e a comunicação já ocorre, o sistema começa a treinar a rede neural com os dados que são retornados.
Reset dos sistemas.	Todos os momentos que a nave é destruída pelos inimigos, o sistema é resetado. Isto ocorre no script Lua com os comandos do emulador. O comando utilizado para realizar o reset é “emu.softreset()”

Fonte: Autor (2018).

4.2 Cálculo do fitness

Diversas funções de cálculo foram implementadas e nenhuma funcionou muito bem: este foi o maior desafio do experimento. A função que obteve melhor resultado, e então foi utilizada nos testes, foi a que utilizava o maior *frame* que a nave conseguiu atingir durante uma jogada. Além disso, para cada *frame* era verificado quantos inimigos estavam com distância menor que 10 unidades, se removendo 0,1 para cada nave próxima e 10 para qualquer disparo próximo. A função de *fitness* ficou então desta forma:

$$fitness = maiorframe - (\sum inimigospróximos * 0{,}1 + \sum disparosnimigospróximos * 10)$$

As naves e os disparos inimigos têm pesos diferentes pois contra as naves pode ocorrer deste inimigo ser eliminado, não ocorrendo o mesmo para os disparos, uma vez que estes não podem ser destruídos, fazendo com que o personagem precise necessariamente se desviar.

4.3 Episódios

Um episódio pode ser considerado uma tentativa de uma população de tentar suceder. Para cada tentativa, um indivíduo é selecionado da população e então as entradas e saídas são proporcionadas pela rede. Após falha ou sucesso desse indivíduo é então cálculado o seu *fitness* e então é avaliado a forma de evolução.

4.4 Resultados

Os resultados obtidos a partir de treinamentos utilizando variações de gerações e populações foram os seguintes:

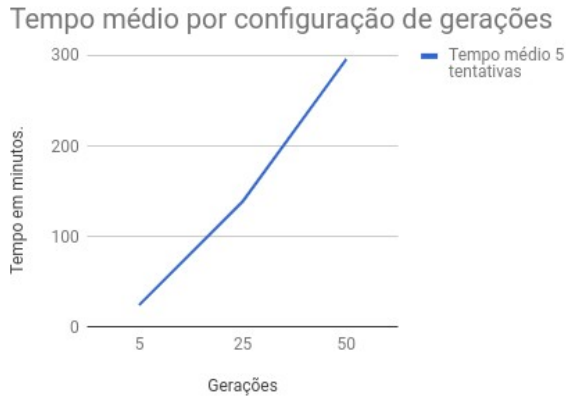
Quadro 7: Tabela de resultados dos treinamentos da rede neural.

Gerações	População	Tempo (min) médio 5 tentativas	Tempo total(min)	Fit médio de 5 tentativas	fit_max	frame_max
5	5	1	3	2748	2858	2317
5	20	4	19	2618	2950	2432
5	50	10	48	2917	3044	2433
5	100	4	22	2864	3045	2410
25	5	17	83	3110	3191	2436
25	20	34	169	3975	5059	4834
25	50	7	36	2985	3105	2433
25	100	42	211	3550	4187	3411
50	5	83	417	4033	5429	4831
50	20	13	63	3043	3164	2435
50	50	77	383	3323	3516	2704
50	100	167	848	4458	6059	5327

Fonte: Autor (2018).

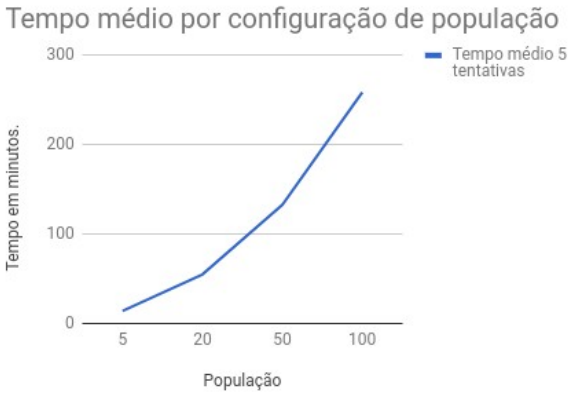
A primeira coisa que pode se analisar é que a evolução é ligeiramente exponencial no tempo para realizar o treinamento da rede neural, apresentados nas figuras 13 e 14:

Figura 13: Gráfico demonstrando evolução de tempo com a variação de gerações.



Fonte: Autor (2018).

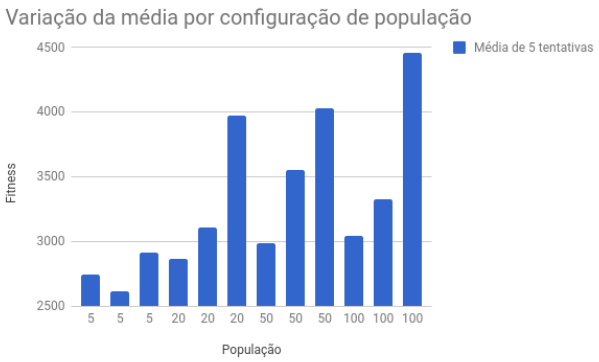
Figura 14: Gráfico demonstrando evolução de tempo com a variação de população.



Fonte: Autor (2018).

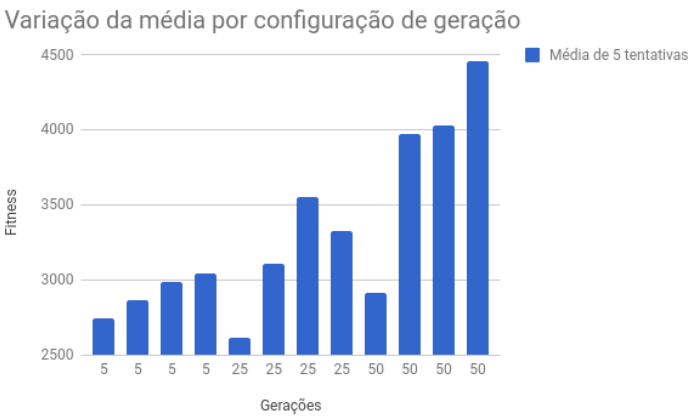
Nas figuras 15 e 16 pode-se perceber uma evolução significativa quando alternado entre número de gerações e população.

Figura 16: Variação de fitness com diferentes configurações de populações.



Fonte: Autor (2018).

Figura 15: Gráfico demonstrando a variação de fitness com diferentes configurações de gerações.



Fonte: Autor (2018).

Durante os treinamentos, foi possível perceber que a rede neural encontrava lugares onde não era possível a nave Vic Viper ser acertada e com isso conseguia grandes pontuações apenas ficando sem sem movimentar para baixo e para cima no canto da tela. Esses casos garantem um *fitness* grande pois o maior fator para o fitness é ficar vivo. As próximas imagens demonstram isso ocorrendo.

Figura 17: Imagem demonstrando a nave parada em uma posição onde não é atingida.

Fonte: Autor (2018).

Figura 18: Outra imagem demonstrando a nave parada. Desta vez quase no final do cenário.

Fonte: Autor (2018).

5 CONCLUSÕES

Nesta monografia foi realizada a análise do algoritmo de aprendizado NEAT aplicado a um jogo eletrônico do Nintendo Entertainment System (NES), utilizando emulador específico para este console, comunicação entre processos em Lua e rotinas de Aprendizado por Reforço escritas em Python.

A utilização do algoritmo NEAT apresentou-se satisfatória para treinamento de uma rede neural, demonstrando resultados muitos bons para pouco tempo de treinamento e considerando que o jogo Gradius é um jogo onde é necessário movimentação constante e destruição das naves para conseguir sobreviver

Durante as realizações das simulações foi possível perceber que a evolução ocorria de forma gradual, começando com a nave fazendo nada e se destruindo diversas vezes para os primeiros inimigos para terminar com ela descobrindo posições em que poderia ficar sem ser alvejada. Também foi possível perceber que a nave dava prioridade em alguns momentos para destruição das naves inimigas, desta forma podendo obter mais pontuação e sobrevivendo durante mais tempo.

Um dos desafios foi o entendimento do pacote NEAT-Python, pois dependia de compreender as explicações e exemplos que constam na sua documentação. Além disso, o aprendizado de Lua e a comunicação com *pipes* bastante trabalhosos, pois não havia nenhum conhecimento prévio. Além disto, a comunicação através dos *pipes* foi essencial para este trabalho, sendo seu aprendizado bastante complicado.

O código fonte deste trabalho pode ser encontrado no seguinte link do Github:

https://github.com/alanbre/Gradius-NEAT

5.1 Trabalhos futuros

Uma limitação deste trabalho é que não foi possível terminar a primeira fase do jogo Gradius com a quantidade de testes feitos. Acredita-se que, com mais treinamento e ajustes da rede neural, seja possível concluir a fase e atingir o inimigo chefe ao final.

Outras coisas que poderiam ser feitas em futuros trabalhos são: a melhoria da função *fitness*, utilização de GPU para tentativa de melhor desempenho no treinamento da rede neural, combinações diferentes de populações e gerações ou ainda configuraçẽos diferentes para o pacote **NEAT-Python**.

REFERÊNCIAS

ALPAYDIN, Ethem. **Introduction to Machine Learning.** Second Edition. Cambridge: Massachusetts, 2010.

BRAGA, Antônio de Pádua; CARVALHO, André Carlos Ponce de Leon Ferreira de; LUDEMIR, Teresa Bernarda. **Redes Neurais Artificiais:** TEORIA E APLICAÇÕES. 2° edição. 2011.

DATACRYSTAL. Disponível em: <https://datacrystal.romhacking.net/wiki/Gradius> Acessado em 25 de out. de 2017.

DENG, Jia; DONG, Dei; SOCHER, Richard; LI, Li-Jia, LI, Kai; FEI-FEI, LI. **ImageNet:** A Large-Scale Hierachial Image Database. 2009.

HOOJAT, Babak. **The AI Resurgence: why now?.** Mar, 2015 Disponível em: <https://www.wired.com/insights/2015/03/ai-resurgence-now/> Acesso em: 25 out. 2017

JACKSON JR, Philip C.. **Introduction to Artificial Intelligence** - Second, Enlarged Edition. Canadá, 1985.

JONES, M. Tim. **Artificial Intelligence** - A System Approach. Hingham, Massachusetts, 2008.

KRAMER, O. **Genetic Algorithms Essentials.** International publishing, 2017

LECUN, Yann; BOTTOU, Léon; BENGIO, Yoshua; HAFFNER, Patrick. **Gradient-based Learning Applied to Document Recognition.** IEEE, 1998.

MCCALL, John. **Genetic Algorithms For Modelling And Optimisation.** School of Computing, Robert Gordon University. Escócia : United Kingdom. 2005.

MILLINGTON, Ian; FUNGE, John. **artificial intelligence for games.** Second Edition. United Kingdom: Oxford, 2009.

MNIH, Volodymyr; KAVUKCUOGLU, Koray; SILVER, David; GRAVES, Alex; ANTONOGLOU, Ioannis; WIERSTRA, Daan; RIEDMILLER, Martin. **Playing Atari with Deep Reinforcement Learning**, DeepMind Technologies. 2013.

OPENAI. **Dota 2.** 2017. Disponível em: <https://blog.openai.com/dota-2/> Acessado em Junho de 2018.

PRESS, Gil. **A Very Short History Of Artificial Intelligence (AI).** 30 dez, 2016. Disponível em <https://www.forbes.com/sites/gilpress/2016/12/30/a-very-short-history-of-artificial-intelligence-ai/2/#6831f88032dc> Acessado em 25 de out. de 2017.

RITCHIE, Dennis M.; THOMPSON, Ken. T**he Unix Time-sharing System.** Bell Laboratories, 1974.

RUSSEL, Stuart J; NORVIG, Peter. **Artificial Intelligence: A Modern Approach**. Third Edition. Upper Saddle River, New Jersey, 2010.

SAS. **Machine Learning. Oque é e qual a sua importância.** 2018. Disponível em <https://www.sas.com/pt_br/insights/analytics/machine-learning.html> Acessado em junho de 2018.

SILVER, David; SCHRITTWIESER, Julian; SIMONYAN, Karen. **Mastering the Game of Go Without Human Knowledge.** Deepmind, 2017.

SMOLA, Alex; VISHWANATHAN S.V.N. **introduction to machine learning**. Departments of Statistics and Computer Science, Purdue University, 2008.

STANLEY, Kenneth O.; MIIKKULAINEN, Risto. **Evolving Neural Netoworks Through Augmenting Topologies**. Department of Computer Science, University of Texas, Austin. 2002.

SUTTON, Richard S.; BARTO, Andrew G. **Reinforcement Learning: An Introduction**. Second Edition. Cambridge, Massachusetts. 2016.

TEAHAN, Willian John. **Artificial Intelligence - Agent Behaviour**. First Edition. 2010

TESSLER, Chen; GIVONY, Shahar; ZAHAVY, Tom; MANKOWITZ, Daniel J.; MANNOR, Shie. **A Deep Hierarchical Approach to Lifelong Learning in Minecraft.** Technion Israel Institute of Technology, Haifa, Israel, 2017.

TUTORIALSPOINT. **Genetic Algorithms.** 2017. Disponível em: <https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_crossover.htm> Acessado dia 26 de out. 2017.