Avaliando a técnica de Aprendizado por Reforço NEAT quando aplicada a uma rede neural jogando um videogame de console de 8 bits

Alan Soder[1], Marcelo de Gomensoro Malheiros[2]

Resumo: Junto com a recente popularidade da Inteligência Artificial, amparada na redução dos custos de hardware massivamente paralelo, surgiu um crescente interesse em algoritmos de auto-aprendizado. Em outras palavras, em processos que permitam a uma rede neural aprender de forma não-supervisionada. Quando o ambiente para aprendizado é dinâmico, podem ser aplicadas técnicas de Aprendizado por Reforço, que permitem que uma rede neural seja refinada por sucessivas tentativas. Em particular, ambientes simulados como os providos por jogos de computador são ideais para aplicar este aprendizado, tanto pelo desafio técnico em si como pelo seu fator lúdico. Este artigo descreve um estudo sobre a aplicação do algoritmo NEAT para treinar automaticamente uma rede neural para jogar o clássico videogame de ação Gradius, lançado para o console de 8 bits Nintendo Entertainment System (NES). O algoritmo NEAT é especialmente interessante pois aplica técnicas de algoritmos genéticos para evoluir a topologia de uma rede neural, modificando também os pesos das suas conexões. Este trabalho fundamenta as questões técnicas envolvidas, apresentando os resultados da ligação entre uma rede neural que se inicia com zero conhecimento e um emulador capaz de executar o jogo eletrônico. São analisados os diversos efeitos dos parâmetros de aprendizado e dos ajustes feitos na função de *fitness*. Ao final, é demonstrado que a rede neural artificial se torna capaz de jogar autonomamente um trecho inicial daquele jogo.

Palavras-chave: Inteligência Artificial. Aprendizado de Máquina. Aprendizado por Reforço. NEAT. Jogos Eletrônicos.

1 Graduado em Engenharia da Computação pela Universidade do Vale do Taquari – UNIVATES. alanbr4@gmail.com

2 Professor na Universidade do Vale do Taquari – UNIVATES. Doutor em Computação na UFRGS – RS, mestre em Engenharia Elétrica na UNICAMP – SP, graduado em Engenharia da Computação na UNICAMP – SP. mgm@univates.br

1 INTRODUCÃO

A Inteligência Artificial (IA) é um dos campos mais novos na área da Ciência e da Engenharia, tendo início após a segunda guerra mundial. 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).

De acordo com Hoojat (2015), 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 geralmente 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.

Ainda que a área da Inteligência Artificial esteja em evidência, os avanços mais impressionantes dos últimos anos estão ligados a Redes Neurais Artificiais (RNA) e a resultados expressivos de precisão no reconhecimento de padrões, muitas vezes ultrapassando humanos em tarefas como tradução de texto, reconhecimento de imagens ou análises não-triviais de indicadores biológicos. A sub-área da IA que se dedica ao treinamento e avaliação de redes neurais é chamada de Aprendizado de Máquina.

Este trabalho pretende estudar e avaliar um caso particular de treinamento de rede neural, quando não há intervenção humana e quando os dados para aprendizado provêm de um processo de retroalimentação automatizado. Nesse caso, estamos falando do tipo de treinamento conhecido como Aprendizado por Reforço (do inglês Reinforcement Learning).

Foi selecionado como objeto de estudo o algoritmo genético NeuroEvolution of Augmenting Topologies (NEAT), que emprega técnicas de modificação evolutiva para maximizar os resultados da aprendizagem. Como ambiente de simulação para prover as medidas de treinamento, escolheu-se o jogo Gradius, título clássico do console de videogame de 8 bits Nintendo Entertainment System (NES).

Este artigo descreve como um jogo pode ser controlado por uma rede neural artificial, funcionando dentro de um software emulador especializado. 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 ligado a um sistema de emulação do console NES, e finalmente avaliar os resultados obtidos.

2 REFERENCIAL TEÓRICO

Esta seção define brevemente os conceitos principais utilizados neste trabalho, assim como dá uma visão geral sobre o funcionamento do algoritmo NEAT.

2.1 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).

2.2 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.

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).

2.3 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.

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 neural de AR refletem em futuros eventos (SUTTON; BARTO, 2016).

2.4 Algoritmo Genético

Um Algoritmo Genético (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 à 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.

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. A função de aptidão (em inglês, *fitness*) é utilizada para medir a qualidade de um cromossomo como solução para um problema específico.

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.

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. 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.

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).

2.5 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.

NEAT é designado para que 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.

Em NEAT, a mutação pode ocorrer de duas formas: 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. 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).

3 DESENVOLVIMENTO

Foi desenvolvido um sistema para avaliação do algoritmo NEAT aplicado ao jogo eletrônico Gradius do Nintendo Entertainment System. Isso foi feito através de *scripts* escritos nas linguagens Lua e Python, utilizando emulação através do emulador FCEUX.

O sistema operou e foi avaliado em um computador pessoal do tipo *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 120GB com velocidades de leitura de 550 MB/s e escrita 500 MB/s. O sistema operacional utilizado foi o Ubuntu versão 16.04.

O jogo Gradius foi executado pelo emulador, FCEUX. Informações importantes sobre o estado do jogo foram extraídas diretamente da memória do sistema emulado, incluindo quantidade de vidas do jogo, se o jogador está vivo ou não, posição vertical e horizontal da nave do jogador, dados sobre os inimigos e pontuação.

3.1 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*.

A execução do jogo Gradius no computador ocorreu através do 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.

Gradius é um jogo do estilo *shoot'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.

3.1.1 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. As informações obtidas da memória do emulador estão no Quadro 1.

Quadro 1: Informações extraídas com o *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: Elaborado pelos autores (2019).

Todas essas informações são obtidas através do *script* Lua funcionando dentro do próprio emulador. Para a realização deste trabalho, os comandos do emulador utilizados estão listados no Quadro 2.

Quadro 2: Lista de comandos utilizados.

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: Elaborado pelos autores (2019).

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 ilustrado no Quadro 3.

Quadro 3: Exemplo de função escrita em Lua.

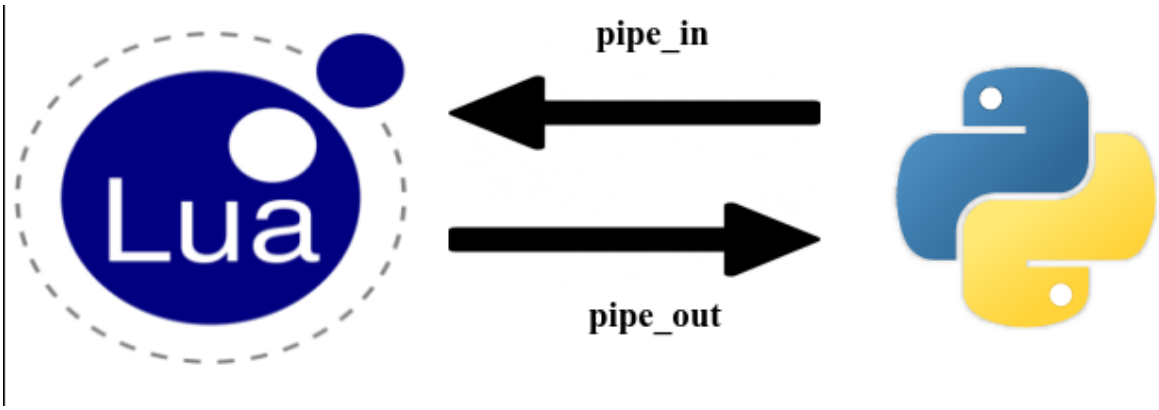
```
function write_to_pipe(data)
    pipe_out, _, _ = io.open(pipe_in_path, "w");
    if data and pipe_out then
        pipe_out:write(data);
        pipe_out:flush();
        pipe_out:close();
    end;
    return;
end;
```

Fonte: Elaborado pelos autores (2019).

3.1.2 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 (RITCHIE; THOMPSON, 1974). O esquema de comunicação utilizado no trabalho pode ser visto na Figura 1. Como em um pipe há apenas um caminho de comunicação, há necessidade de dois para haver ida e volta de informação.

Figura 1: Esquema de comunicação entre as duas linguagens.



Fonte: Elaborado pelos autores (2019).

3.1.3 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.

3.1.4 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 biblioteca também possui funções de controle de crescimento da rede neural, aplicando o conceito 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.

3.2 Sistemática experimental

Para realizar um treinamento de uma rede neural utilizando o pacote NEAT-Python foram definidos os parâmetros mostrados no Quadro 4, configurados no arquivo de configuração da rede.

Quadro 4: 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: Elaborado pelos autores (2019).

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, listados no Quadro 5.

Quadro 5: Lista 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: Elaborado pelos autores (2019).

3.3 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. 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 calculado o seu *fitness* e então é avaliado a forma de evolução.

4 ANÁLISE DOS RESULTADOS

Os resultados obtidos a partir de treinamentos utilizando variações de gerações e populações estão listados no Quadro 6.

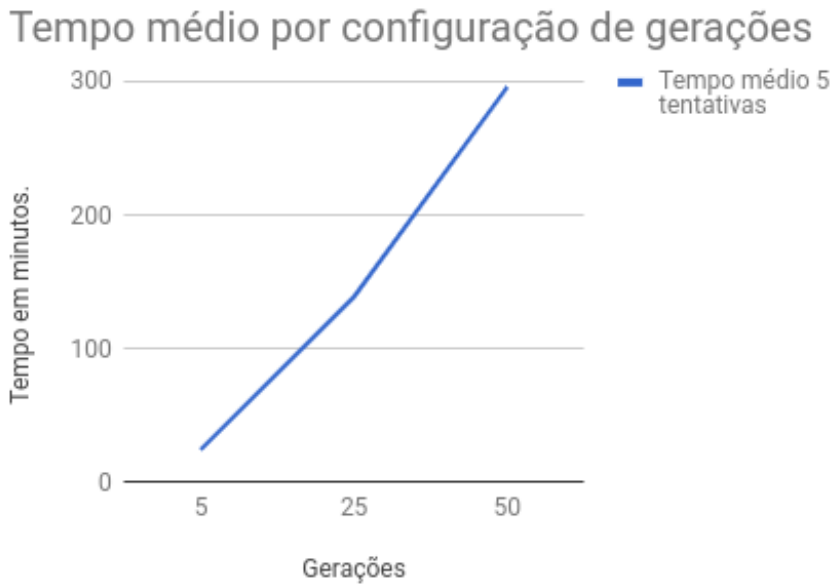
Quadro 6: Medidas após a variação de diversos parâmetros de treinamento.

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

Elaborado pelos autores (2019).

Inicialmente observa-se que a evolução segue uma tendência exponencial no tempo para realizar o treinamento da rede neural, tal como apresentado nas Figuras 2 e 3:

Figura 2: Gráfico demonstrando evolução de tempo com a variação de gerações.



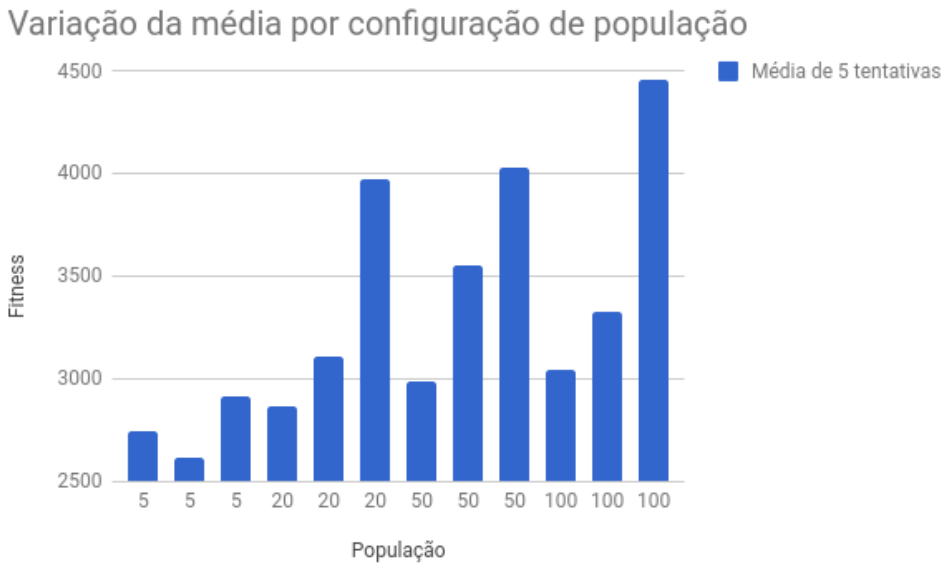
Fonte: Elaborado pelos autores (2019).

Figura 3: Gráfico demonstrando evolução de tempo com a variação de população.

Fonte: Elaborado pelos autores (2019).

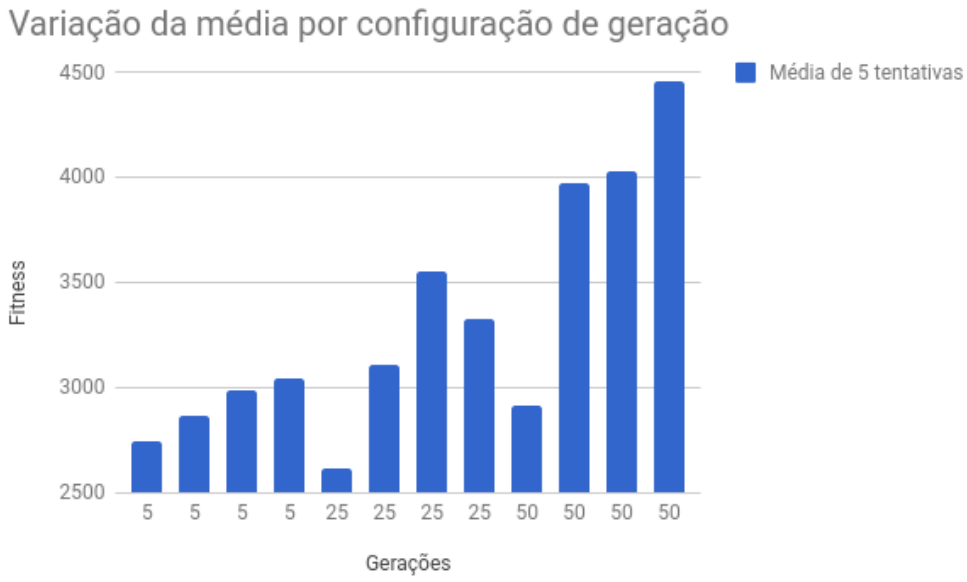
Nas Figuras 4 e 5 pode-se perceber uma evolução significativa quando é variado o número de gerações e da população.

Figura 4: Variação de fitness com diferentes configurações de populações.



Fonte: Elaborado pelos autores (2019).

Figura 5: Variação de *fitness* com diferentes configurações de gerações.



Fonte: Elaborado pelos autores (2019).

Durante os treinamentos foi possível perceber que a rede neural encontrava lugares onde não era possível a nave Vic Viper ser acertada, conseguindo então grandes pontuações sem movimentação. Esses casos garantem um *fitness* grande pois o maior fator para o *fitness* é permanecer vivo. As Figuras 6 e 7 ilustram isso ocorrendo.

Figura 6: Imagem demonstrando a nave parada em uma posição onde não é atingida.

Fonte: Elaborado pelos autores (2019).

Figura 7: Outra imagem demonstrando a nave parada, desta vez quase no final do cenário.

Fonte: Elaborado pelos autores (2019).

5 CONCLUSÕES

Neste trabalho 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.

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 configurações diferentes para o pacote NEAT-Python.

REFERÊNCIAS

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

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

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

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

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.

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.