Evoluindo Redes Neurais para Jogar Mega Man X

Projeto Final de MO444/MC886 – Aprendizado de Máquina

Alexandre E. Almeida	Nathália M. C. dos Santos	Elvis R. C. Quispe	Jose L. F. Campana	Júlia F. Tessler
106535	147512	209813	209820	119655
almeida.xan@gmail.com	nathmenini@gmail.com	e209813@g.unicamp.br	121451@unsaac.edu.pe	juliatessler@gmail.com

I. Resumo

Técnicas de Aprendizado de Máquina, como Redes Neurais, em conjunto com Algoritmos Genéticos têm sido amplamente utilizadas com a finalidade de obter bons resultados em termos de desempenho na área de jogos. Neste trabalho, propomos utilizar o algoritmo *NeuroEvolution of Augmenting Topologies* para vencer a fase Highway Stage do jogo Mega Man X do console Super Nintendo Entertainment System. Para este propósito, informações de cada *frame* do jogo foram utilizadas tanto para o *input* das Redes Neurais quanto para a função *fitness* do Algoritmo Genético. Os resultados mostram que, embora o principal objetivo não tenha sido totalmente atingido, a abordagem proposta se mostrou muito promissora em sua função, evidenciando que outras tentativas e um maior tempo de treinamento poderiam, de fato, produzir o resultado desejado.

II. Introdução

O recente aumento de interesse do público em geral nas áreas de Aprendizado de Máquina (AM) e Inteligência Artificial (IA) nos últimos anos desencadeou melhorias não só na área de pesquisa acadêmica, como também na indústria. AM e IA abrangem uma enorme variedade de subcampos, desde áreas de uso geral como, por exemplo, aprendizado e percepção, até tarefas específicas como jogos de xadrez [1] e demonstração de teoremas matemáticos [2]. Além disso, essa grande área está envolvida em temas como visão computacional [3], análise e síntese da voz [4] e lógica *fuzzy* [5].

Especificamente na indústria de jogos, desenvolvedores têm se beneficiado de técnicas de AM para, por exemplo, realizar geração procedural de conteúdo [6], de modo a aumentar e diversificar o conteúdo disponível em jogos, diminuindo a demanda de trabalho manual e, portanto, reduzindo custos e tempo de produção.

Em contrapartida, as comunidades de jogadores também têm se beneficiado de técnicas de AM e IA. Tomando como exemplo o jogo *multiplayer* Dota 2[1], técnicas de Regressão Logística e Redes Neurais (RN) já foram utilizadas para predizer o time vencedor com base nos personagens escolhidos pelos jogadores[2] ou, mais recentemente, na criação de um *bot* pela empresa OpenAI, o qual foi capaz de vencer os melhores jogadores profissionais do mundo em partidas um-versus-um[3], Um dos trabalhos mais recentes sobre Neat presentado no trabalho

Jogadores mais casuais, no entanto, fazem parte de um nicho mais impulsionado por diversão e curiosidade, utilizando programas puramente para criar IAs que são capazes de obter bons desempenhos em jogos. Com essa finalidade, frequentemente utiliza-se o algoritmo *NeuroEvolution of Augmenting Topologies* (NEAT) [7]. O algoritmo NEAT realiza a evolução de RNs através de um Algoritmo Genético (AG), guiado por uma função de *fitness* que direciona o processo evolutivo. Tal algoritmo já foi utilizado em jogos como Flappy Bird, QWOP e Super Mario World.

Com este trabalho, propomos desenvolver um programa que seja capaz de “ensinar” uma IA a jogar uma das fases do jogo Mega Man X (MMX), do console Super Nintendo Entertainment System (SNES). Para isso, utilizaremos o algoritmo NEAT, mais especificamente uma implementação na linguagem Lua produzida por Seth Bling ao desenvolver uma IA que foi capaz de vencer a fase Donut Plains I do jogo Super Mario World[4], também no SNES.

Uma característica em comum entre as implementações do NEAT em outros jogos foi a escolha de fases lineares, permitindo a definição de uma função de *fitness* simples—quanto mais para a esquerda na fase (início), menor a pontuação, e quanto mais para a direita (final), maior a pontuação. Por esta razão, também escolhemos uma fase com característica linear—a primeira fase do MMX, Highway Stage, ilustrada na Figura 1.

No entanto, enquanto que em outros jogos mais simples a mecânica é limitada apenas por movimentos direcionais, pulos, e um estado vivo/morto do herói, no MMX a mecânica é um pouco mais complexa. Por exemplo, alguns dos inimigos são muito grandes, e obrigatoriamente devem ser derrotados (com tiros) para permitir que o herói dê continuidade na fase. Além disso, o herói do jogo conta com uma barra de vida (esquerda da Figura 1), permitindo que ele suporte alguns ataques antes de morrer (a vida pode ser recuperada ao coletar-se itens que eventualmente caem de inimigos mortos). Todas essas mecânicas adicionam graus de complexidade que devem ser levados em conta ao construir-se a função de *fitness* do AG. Um dos desafios ao utilizar-se o NEAT é encontrar indivíduos

[1] Dota 2, Valve *Corporation*. http://dota2.com [Acessado em 11/2017].

[2] *Tool that predicts the outcome of a Dota 2 game using machine learning.* https://github.com/andreiapostoae/dota2-predictor [Acessado em 11/2017].

[3] OpenAI + Dota 2. https://blog.openai.com/dota-2/ [Acessado em 11/2017].

[4] MarI/O - ML for Video Games. https://youtu.be/qv6UVOQ0F44 [Acessado em 11/2017].

Figura 1: Início da fase Highway Stage do jogo MMX.

que possuem alta pontuação para a função de *fitness* e, durante a busca por tais indivíduos, o algoritmo muitas vezes é vítima de máximos locais e tem dificuldades em encontrar melhores soluções [8].

III. Trabalhos Relacionados

Existem diversos trabalhos que evidenciam o grande interesse e esforço para ensinar uma IA a jogar um jogo. O *YouTuber* Seth Bling, por exemplo, além de ensinar uma IA que foi capaz de vencer a fase Donut Plains I do jogo Super Mario World utilizando o algoritmo NEAT, recentemente utilizou Redes Neurais Recorrentes para ensinar uma outra IA a ganhar diversas fases do jogo Mario Kart[5]. O NeatMonster, outro canal no *YouTube*, utilizou o NEAT para ensinar uma IA a jogar Flappy Bird[6].

Além disso, Foley e Kuhn [9], utilizaram o algoritmo NEAT e EBTs (*Evolving Behaviour Trees*) para completar uma fase do jogo Super Mario Bros. (do console Nintento Entertainment System, NES) com o objetivo de comparar a performance desses dois algoritmos. Já em [10], os autores utilizaram o NEAT e o sistema classificador de aprendizado baseado em precisão para ensinar uma IA a jogar o jogo Robocode[7].

Boris e Goran [11] também utilizaram o algoritmo NEAT para treinar RNs para aprender a jogar o jogo 2048vv, um jogo do tipo *puzzle* em que o jogador busca alcançar a soma 2048 através de movimentos do tipo cima-baixo e esquerda-direita em um tabuleiro 4x4. Apesar de grandes esforços e, de altas pontuações obtidas pelos autores, os mesmos acreditam que não conseguiram alcançar os melhores resultados com a metodologia que propuseram.

Andersen et al. [12] utilizaram o NEAT para aprender a jogar o jogo de tabuleiro Othello, também conhecido como Reversi. Othello é um jogo de dois jogadores, em que o objetivo é popular o espaço do tabuleiro com suas peças. Os autores treinaram as redes com duas estratégias de jogo: focada em posicionamento das peças e em coevolução competitiva. Com isso, conseguiram que a rede aprendesse a se adaptar aos movimentos do oponente.

Hausknecht et al. [13] propuseram a utilização do NEAT e de outros algoritmos como, por exemplo, o HyperNEAT [14] para ensinar uma RN a jogar alguns jogos do console Atari 2600.

Uma nova versão do NEAT aparece em [15], chamada *real-time NeuroEvolution of Augmenting Topologies* (rtNEAT). A proposta dos autores é um algoritmo que evolui, em tempo real, RNs cada vez mais complexas durante um jogo. Desta forma, os autores introduzem uma nova forma de se construir jogos, em que as características do mesmo mudam dependendo da forma com que se joga. Por fim, para demonstrar o poder do algoritmo, é introduzido um jogo chamado *NeuroEvolving Robotic Operatives* (NERO), baseado em conceitos do rtNEAT, em que um jogador treina um time de robôs para lutar contra outros times. Essa aplicação é consideravelmente diferente do que propomos neste trabalho, mas demonstra o poder dos algoritmos NEAT aplicados a jogos virtuais. Trabalhos mais recentes, como em [16], sinalizam que a utilização do NEAT pode ser estendida para o contexto de Aprendizado Profundo, em que os autores propõem o *Coevolution* DeepNEAT, que é uma versão do NEAT que buscar otimizar arquiteturas de aprendizagem profunda através da evolução.

IV. Conceitos Básicos

A. Algoritmo Genético

AGs são tipicamente representados por *strings* binárias, entretanto, existem outras variedades de representação disponíveis [17]. O processo evolucionário começa com *indivíduos* aleatórios (também conhecidos como *cromossomos*). A coleção de indivíduos é chamada de *população*. Cada *geração* consiste de uma população de indivíduos. Existem quatro operações básicas que permitem avançar de uma geração para a outra:

- **Avaliação.** Cada indivíduo é avaliado utilizando uma função de *fitness* ao final de sua execução. Essa função atribui uma pontuação numérica para cada indivíduo, tornando possível detectar aqueles que melhor se adéquam à resolução do problema-alvo;
- **Seleção.** No final de cada iteração do algoritmo, os indivíduos com maior pontuação de *fitness* serão selecionados para seguir adiante e, através das operações de mutação e *crossover*, criarão novos indivíduos para as próximas gerações;
- **Mutação.** Para adicionar mais variabilidade a cada geração, os indivíduos sofrem, ocasionalmente, *mutações*, ou seja, algumas informações dos indivíduos são aleatoriamente alteradas. Após ocorrer uma mutação, certos indivíduos poderão aleatoriamente realizar *crossover* com outro indivíduo, difundindo assim a nova informação para a população;
- ***Crossover*** *Crossover* é a ação de trocar porções da informação entre dois indivíduos (pais) da população

[5]Canal de Seth Bling no *YouTube*. https://youtu.be/Ipi40cb_RsI [Acessado em 11/2017].

[6]Link do vídeo do Flappy Bird. https://youtu.be/pEtM1Coktio [Acessado em 11/2017].

[7]Página do jogo Robocode. http://robocode.sourceforge.net/ [Acessado em 12/2017].

para gerar-se um novo indivíduo (filho). O objetivo do *crossover* é criar filhos que são melhores do que seus pais. O *crossover* por si só nem sempre é suficiente para criar mudanças significativas de uma geração para a outra e, por este motivo, somente indivíduos que são mutados sofrem *crossover*.

Usualmente, os AGs são executados de forma contínua, convergindo lentamente para a solução ótima. Entretanto, por padrão, os AGs nunca param automaticamente, pois necessitam de uma condição de parada. Para isso, pode-se, por exemplo, incluir um limitador de pontuação de *fitness*, um número máximo de gerações ou um tempo máximo de processamento.

B. Redes Neurais

Uma RN é uma técnica de IA inspirada na maneira como o nosso cérebro funciona [18]. Uma RN consiste em um conjunto de neurônios que são interligados através de conexões com pesos. De maneira geral, os neurônios podem ser do tipo *input*, *output* ou *hidden*. Na Figura 2 apresentamos a estrutura geral de uma RN *feed-forward*, cujos principais conceitos são explicados a seguir.

- ***Input.*** Esses neurônios recebem valores relacionados a entrada do modelo que está sendo treinado;
- ***Output.*** Os neurônios do *output* representam a saída do modelo, que podem ser, por exemplo, uma predição ou uma classificação, dependendo do objetivo;
- ***Hidden.*** Os neurônios da *hidden* são utilizados para adicionar mais detalhes para a saída da RN. Se o *output* de uma RN original foi representado por uma função $f(x)$, então adicionando a camada de neurônios da *hidden*, devemos ter no *output* $f(g(x))$, em que $g(x)$ representa a função de ativação da camada *hidden*.

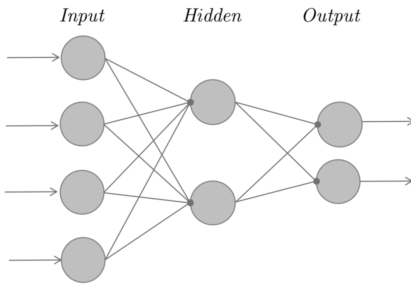


Figura 2: Arquitetura geral de uma RN *feed-forward*.

Mais especificamente, a saída de cada neurônio é determinado por seus *inputs* e pela sua função de ativação [19]. O valor do *input* para um neurônio j (in_j) é determinado pelo valor de cada um de seus *inputs* (x_i) e pelo seus pesos (w_i), ou seja,

$$in_j = \sum_i w_i x_i. \quad (1)$$

A saída do neurônio é, então, computada utilizando a função de ativação ($g(x)$), ou seja,

$$out_j = g(in_j). \quad (2)$$

Existem muitas possibilidades para a função de ativação. Uma das funções mais básicas é aquela que atribui o valor 1 ao *output* do neurônio se $in_j \geq 0$ e 0 caso contrário. Uma função mais complexa pode ser uma função diferenciável, como a função sigmóide

$$g(in_j) = \frac{1}{1 + e^{\frac{in_j}{p}}}, \quad p \in \mathbb{R}. \quad (3)$$

A etapa de estimação dos pesos (w_i) se dá através do algoritmo de *backpropagation* [18], [19].

C. Algoritmo NEAT

A Aprendizagem por Reforço (AR), uma das áreas de pesquisa mais ativas na IA, é uma abordagem computacional para aprendizagem, na qual um agente tenta maximizar a quantidade total de recompensa recebida ao interagir com um ambiente complexo e incerto [20]. No caso de um jogo, a personagem principal pode ser vista como o agente que faz ações e, com isso, muda os estados (ou configurações) do jogo —o que faz com que o ambiente seja complexo e incerto—, e a função de *fitness* é responsável por mensurar a recompensa recebida.

Neuroevolution (NE), a evolução artificial das RNs utilizando AG, apresentou grande promessa em tarefas complexas na área de AR [21]–[24]. Nesse caminho, uma ténica de AR que se mostrou extremamente eficiente é o NEAT [7]. Basicamente, o NEAT combina AG com RN para criar uma técnica que busca encontrar a topologia (arquitetura) correta de uma RN.

1) Estrutura Básica: O NEAT codifica a RN utilizando o conceito de genes no AG [7]. Define-se dois tipos de genes:

- **Genes Nó.** Genes nó descrevem um neurônio na RN. Isso inclui os neurônios da camada *input*, *output* e *hidden*;
- **Genes Conexão.** Os genes de conexão descrevem a conexão entre dois genes nó. Um gene conexão inclui informação sobre o nó de entrada, nó de saída, o peso da conexão, *enable bit* e um número de inovação que ajuda a identificar a origem no gene conexão [7].

2) Mutação: O NEAT permite que quase todas as partes da RN sofram mutação. Assim como diversas técnicas de aprendizado, o NEAT permite que o peso do gene conexão mude. Entretanto, o NEAT também permite que a topologia da RN mude através de *crossover* e mutação. Quando adicionamos um novo gene nó, um gene conexão que existia é dividido em dois. Um deles terá o valor do peso antigo e o outro terá peso igual a 1. Isso garante que o novo nó não altere nenhum cálculo imediatamente [7]. Ao adicionar um gene conexão, o seu peso é gerado aleatoriamente e, assim, obtém-se um novo número de inovação.

3) Número de Inovação: Como descrito anteriormente, os genes conexão possuem um número de inovação que ajuda a identificar quando um gene foi originalmente criado [7]. Desse modo, tem-se uma maneira fácil de identificar a origem do gene, o que ajuda a evitar a perda ou duplicação de genes conexão durante o treinamento.

4) Crossover: A cada geração, todas as topologias dos indivíduos são avaliadas em relação ao objetivo para a qual elas foram projetadas. Para cada uma, é atribuída uma pontuação de *fitness* baseado em quão bem elas se apresentaram. Aquelas com as maiores pontuações de *fitness* são mais propensas de serem selecionadas para realizar *crossover*. O processo de *crossover* do NEAT é ilustrado na Figura 3. Primeiro, números de inovação são usados para fazer a correspondência dos genes dos dois pais [7]. Os genes conexão que possuem o mesmo número de inovação sofrem *crossover*, de modo que é feita uma média de seus pesos para criar um gene de conexão para o seu sucessor (filho). Em seguida, os genes restantes são classificados como *disjoint* ou *excess*. Os genes *disjoint* têm um número de inovação dentro do alcance dos números de inovação dos outros pais. Os genes *excess* têm um número de inovação fora desse intervalo. Finalmente, os genes *disjoint* e *excess* do pai com a maior pontuação de *fitness* são adicionados ao genoma do filho.

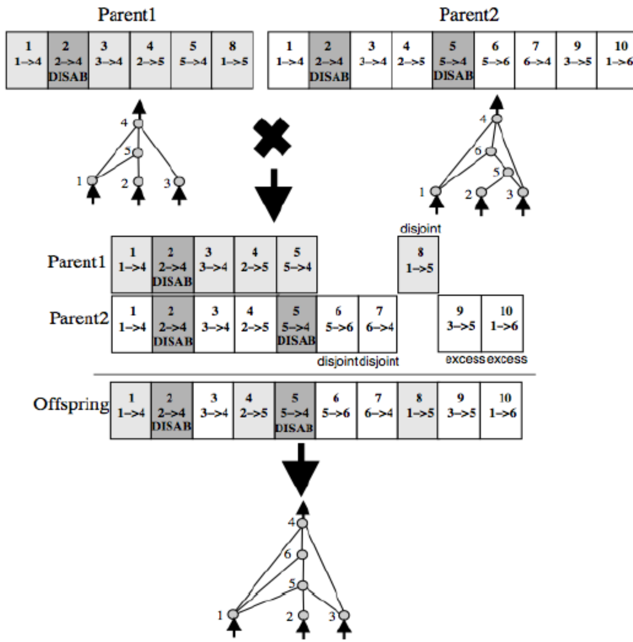


Figura 3: Processo de *crossover* do NEAT. Figura extraída de [7].

5) Speciation: Ao adicionar novos genes a um genoma, há uma chance de que pontuação da função *fitness* da RN caia o suficiente para que ela não sobreviva até a próxima geração. A fim de dar a cada novo gene uma chance justa de mostrar seu real potencial, o NEAT divide a população em espécies [7]. As espécies são criadas agrupando a população com base na distância de compatibilidade δ. Essa distância, conforme descrita em [7], é uma combinação linear do número de genes *excess* (E), o número de genes *disjoint* (D) e a média de diferença dos pesos dos genes correspondentes ($\overline{W}$)

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \overline{W}, \tag{4}$$

em que os pesos c_1, c_2 e c_3 podem ser modificados para alterar o peso de cada um dos três termos e N é o tamanho do maior genoma.

O NEAT usa compartilhamento de *fitness* explícito para garantir que uma espécie não se torne muito grande [7]. Portanto, o novo *fitness* de um organismo (f'_i) é determinado pela equação

$$f'_i = \frac{f_i}{N_s}, \tag{5}$$

em que f_i é o *fitness* original e N_s é o número de organismos na espécie [7]. O objetivo dessa expressão é dar uma maior pontuação para espécies menores e uma menor pontuação para espécies densamente povoadas. De acordo com Kenneth e Miikkulainen [7], o efeito desejado de realizar *speciation* é proteger inovações topológicas e, portanto, o objetivo final do sistema é realizar a busca por uma solução da maneira mais eficiente possível. Isso é possível pois, ao proteger a inovação topológica, o NEAT minimiza a quantidade de trabalho necessário para encontrar uma solução eficiente. O agrupamento por espécies é ilustrado na Figura 4.

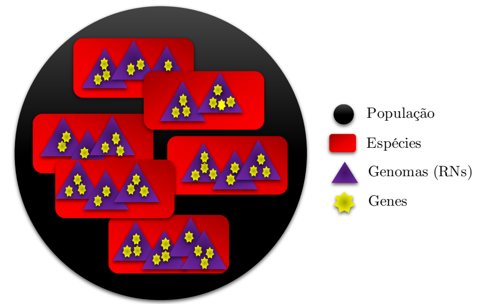


Figura 4: Agrupamento das espécies no algoritmo NEAT.

V. Metodologia

A. Emulador BizHawk

Para rodar o jogo sem o console físico do SNES, utilizamos o emulador BizHawk v2.2[8]. O BizHawk é um emulador projetado predominantemente em torno da produção de *Tool-Assisted Speedruns* (TAS), o qual foi escrito em C#. Esse emulador foi escolhido pois oferece suporte para *scripts* em Lua. Ao ser carregado pelo emulador, o arquivo contendo o jogo tem sua memória mapeada em ROM (memória não-volátil), RAM (memória volátil) e VRAM (vídeo-RAM).

Portanto, devido a limitações do próprio emulador, foi necessário que todas as implementações fossem escritas na linguagem Lua. Como ponto de partida, utilizamos e adaptamos uma implementação do algoritmo NEAT disponibilizada pelo autor Seth Bling[9].

[8]BizHawk *Emulator*. http://tasvideos.org/Bizhawk.html [Acessado em 11/2017].

[9]*Script* em Lua criado por Seth Bling para o jogo Super Mario World. https://pastebin.com/ZZmSNaHX [Acessado em 12/2017].

B. Conjunto de Dados

O conjunto de dados utilizado neste trabalho é coletado à medida em que o jogo avança, ou seja, em tempo real para cada *frame* do jogo. As características voláteis do jogo ficam armazenadas na memória RAM, o que nos permite facilmente acessá-las e minerar informações importantes do jogo. Para algumas das informações do jogo, utilizamos um mapeamento de RAM previamente feito por membros da comunidade do BizHawk, o qual foi disponibilizado pelo TASVideos[10]. Outras informações que não estavam descritas nesse mapeamento, como a localização na memória da informação sobre os blocos sólidos da fase, foi necessário que nós mesmos fizéssemos o mapeamento da RAM, o que nos consumiu consideravelmente muito tempo. Dentre as informações coletadas, temos: a posição dos inimigos, dos tiros dos inimigos e os blocos sólidos nos quais o Mega Man pode andar/escalar.

Na Figura 5, ilustramos os conceitos de *scene*, *block*, *tile* e *pixel*, que são diferentes unidades para as quais informações de blocos da fase são armazenadas na memória RAM. A maior unidade é a *scene*, que é formada por 256×256 px, entretando, para exibir na tela do jogo, a *scene* é cortada para 256×224 px. Em seguida, a *scene* é divida em 8×8 *blocks*, em que cada *block* é formado por 32×32 px, e, ao jogador, são exibidos 8×7 *blocks*. Por último, cada *block* é dividido por 2×2 *tiles*, em que cada *tile* é formado por 16×16 px e, novamente, ao jogador, são exibidos 16×14 *tiles*. A informações da solidez dos blocos ficam armazenadas na memória RAM apenas para *tiles* e, desse modo, todo o mapeamento foi feito considerando essa unidade.

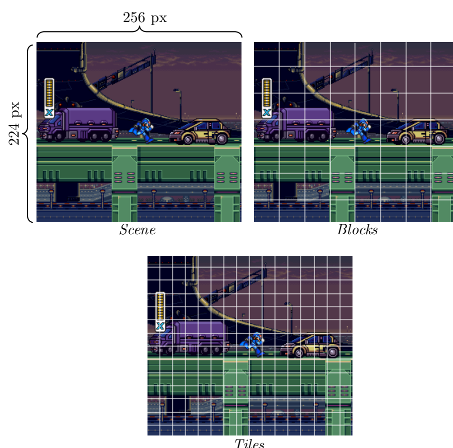


Figura 5: Ilustração das unidades para as quais informações de blocos da fase são armazenados na memória RAM.

Na Figura 6, ilustramos o minimapa que representa uma versão simplificada da tela do jogo. O minimapa foi construído considerando as informações dos *tiles*, inimigos e seus tiros para cada *frame* do jogo, levando em consideração apenas o que é exibido na tela para o jogador. Desse modo, ele é representado por um mosaico de 16×14 quadrados (que representam os 16×14 *tiles* visíveis ao jogador). Para cada valor lido da memória para a construção do minimapa, realizou-se um mapeamento a um valor inteiro. Os *tiles* referentes aos inimigos e seus tiros receberam valor -1 (quadrados pretos), blocos sólidos receberam o valor 1 (quadrados brancos) e, os demais *tiles* que não se enquadravam nessas condições receberam o valor 0 (quadrados vazios). Apenas para fins visuais, também incluímos no minimapa a posição atual do Mega Man (quadrado azul).

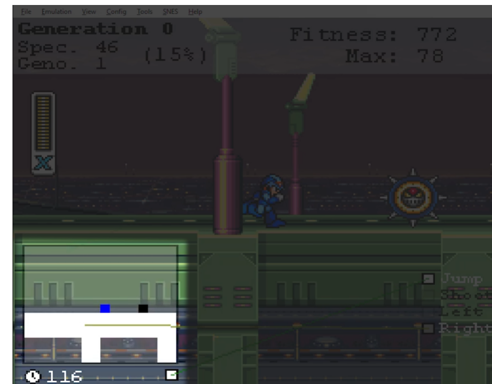


Figura 6: Minimapa do jogo.

Além disso, para compor a função de *fitness*, informações sobre a posição e vida atual do Mega Man e a quantidade de tiros efetivos (que atingiram os inimigos) também foram extraídas da memória RAM a cada *frame* do jogo.

C. Configurações

1) Redes Neurais: Levando em consideração a Seção IV, temos que as RNs são compostas por neurônios de três tipos: neurônios de entrada, de saída e neurônios escondidos. No caso do MMX, a entrada é representada por uma versão vetorizada do minimapa exibido na Figura 6. Já no *output*, temos os botões que serão pressionados: pular, atirar e andar para a direita ou esquerda. Por exemplo, se o neurônio de *output* que representa o botão de pular assumir o valor 1, então a IA deve pressionar o botão de pular. Além disso, cada neurônio pode sofrer mutações e, com isso, dar origem a outro gene nó e gene conexão, sendo que limitamos o máximo número de genes nós em 10000.

Na Figura 7 apresentamos um exemplo da arquitetura da uma RN no NEAT. Nas redes, temos que as conexões de todos os nós da rede são representadas pelas arestas. Nós brancos indicam que o neurônio passa informação positiva, nós pretos indicam informação negativa, e nós transparentes não estão ligados. Arestas verdes indicam passagem de informação com o mesmo sinal, enquanto arestas vermelhas trocam o sinal. Já arestas amarelas não passam informação. A direita temos os nós de *output*, indicando as teclas que serão apertadas. O quadrado branco abaixo do minimapa representa o *bias*.

2) NEAT: A população inicial foi configurada para 300 indivíduos em que cada indivíduo é uma RN. Cada RN é inicializada com pesos aleatórios.

[10] http://tasvideos.org/GameResources/SNES/MegaManX/RAMMap.html [Acessado em 11/2017].

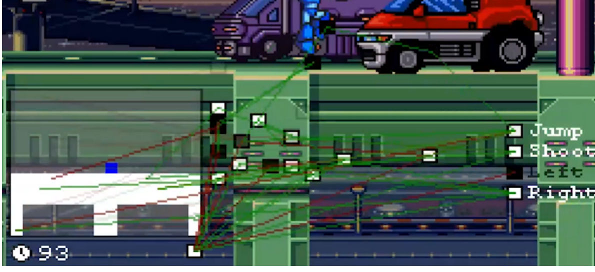


Figura 7: Exemplo da arquitetura da uma RN no NEAT.

Na Tabela I, apresentamos duas configurações dos parâmetros do NEAT utilizados como, por exemplo, a chance de ocorrer a mutação em um indivíduo ou o *crossover* entre dois indivíduos.

Tabela I: Duas configurações dos parâmetros utilizados no algoritmo NEAT

	Configuração	
Parâmetro	I	II
c_1	2.0	1.0
c_2	2.0	2.0
c_3	0.4	0.4
Chance de mutar o gene conexão	2.0	2.0
Chance de mutar o gene nó	0.5	1.0
Chance de mutar o *bias*	0.4	0.8
Chance de *crossover*	0.75	0.75
Chance de habilitar mutação	0.2	0.4
Chance de desabilitar mutação	0.4	0.4

D. Função de Fitness

Quando um indivíduo começa a jogar, ele passa a tomar ações no jogo (como pular, atirar e andar para a direita e esquerda) em resposta aos *inputs* presentes no minimapa, e retorna um novo estado (configuração do jogo) e uma recompensa. A recompensa é mensurada através de uma função de *fitness*. Essa função atribui um valor numérico para cada indivíduo, tornando possível a comparação da performance de indivíduos distintos.

O objetivo é maximizar a função de *fitness* obtida pelos indivíduos, em termos de chegar ao final da fase com êxito. Devido a complexidade do jogo MMX, é necessário definir bem essa função. Para isso, é preciso definir os parâmetros que irão avaliar recompensas imediatas recebidas pelos indivíduos. Também é importante definir uma rotina de *time out* que encerra prematuramente indivíduos que não estão executando ações que aumentem sua função de *fitness* em um certo intervalo de tempo. Além disso, o *time out* deve ser bonificado levando em conta o tempo de vida dos indivíduos, ou seja, quanto mais tempo um indivíduo estiver vivo, maior é o intervalo de folga que lhe é atribuído no *time out*. O *time out* é mostrado pelo número próximo ao relógio na Figura 6, e é mensurado em número de *frames*.

Por fim, definimos que quando a vida do Mega Man atinge 0 (ou seja, ele morre), então instantaneamente esse indivíduo é encerrado e outro indivíduo passa a ser avaliado em seguida. Essa ação agiliza consideravelmente o tempo da evolução do AG, pois deixa de ser necessário esperar a animação do Mega Man morrer e/ou o tempo de *time out* ser atingido, o que pode levar mais de 5 segundos por indivíduo.

VI. Experimentos e Discussão

Todos os experimentos realizados utilizaram, primeiramente, a configuração I apresentada na Tabela I. No primeiro experimento utilizamos uma função de *fitness* baseada apenas na seguinte equação,

$$f_1 = R - R_o, \tag{6}$$

em que R é posição mais à direita alcançada pelo Mega Man e R_o é usado para transformar a posição inicial do Mega Man em 0. Ou seja, com essa função, estamos levando em consideração apenas se Mega Man consegue avançar para a direita na tela. Com essa função, ele aprendeu que andar para a direita era algo positivo, enquanto que para a esquerda não. Além disso, os indivíduos demoraram muito para aprender a pular buracos e não conseguiram aprender que era necessário matar inimigos.

No segundo experimento, utilizamos uma versão modificada de f_1,

$$f_2 = R - R_o + 20 \cdot S, \tag{7}$$

em que S é calculado com base em quantos tiros do Mega Man atingiram os inimigos. Além disso, foi atribuído um peso 20 à quantidade de tiros certeiros, como uma tentativa de agilizar o processo de aprendizado para matar inimigos. Entretanto, apesar de aprender a matar os inimigos, o Mega Man não foi capaz de aprender que era necessário matá-los rapidamente para permanecer vivo.

No terceiro experimento, consideramos uma modificação da função f_2, adicionando informações sobre a vida do Mega Man. A função desse experimento é dada por

$$f_3 = R - R_o + 20 \cdot S - 150 \cdot (16 - H) \tag{8}$$

em que H é a quantidade de vida do Mega Man, no intervalo discreto $\{0, \ldots, 16\}$. Com essa modificação, foi possível começar a ensinar o Mega Man a matar inimigos rapidamente, entretanto, ao se deparar com o primeiro sub-chefe, ele não foi capaz de ficar parado para matá-lo. Além disso, em algumas situações em que ele conseguia matá-lo, ele não conseguia escalar a grande parede em seguida.

Na Tabela II apresentamos outras funções de *fitness* baseadas em f_3. Essa funções foram utilizadas como uma tentativa de ensinar o Mega Man a matar o sub-chefe mais rapidamente e escalar as grandes paredes.

Tabela II: Outras funções de *fitness* utilizadas no treinamento

f	Expressão Matemática
f_4	$3 \cdot (R - R_o) + 100 \cdot S - 20 \cdot (16 - H)$
f_5	$2 \cdot (R - R_o) + 20 \cdot S - 150 \cdot (16 - H)$

Na Tabela III sumarizamos o máximo valor de R que o melhor indivíduo conseguiu alcançar para cada função de *fitness*. Ressaltamos que o valor mais a direita que pode ser alcançado é 7544, que é referente ao final da fase. Pode-se observar que, conforme discutido anteriormente, à medida em que modificávamos as funções inserindo novas informações e maior complexidade, o Mega Man conseguia avançar mais na fase. Uma exceção sendo o valor de R para f_4 menor do que f_3. O motivo é que com f_3 o Mega Man ignorava o sub-chefe e continuava andando para a direita da tela, sem conseguir avançar (já que é necessário matá-lo primeiro). Já os indivíduos em f_4 conseguiam matar o sub-chefe, o que fez com que o chão caísse e eles ficassem presos no buraco, limitando o seu máximo valor de R enquanto não conseguiam escalar a parede. Das cinco funções avaliadas na configuração I, escolhemos as duas melhores (f_4 e f_5) para testar utilizando a configuração II. Dessas, a que apresentou o melhor resultado foi a função f_5. Na Seção VII apresentaremos os resultados para essas 2 funções com maiores detalhes.

Tabela III: Máximo valor de R atingido utilizando as funções e configurações propostas

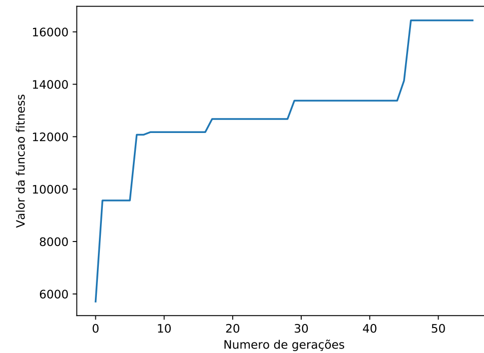
	Máximo valor de R	
Função	Configuração I	Configuração II
f_1	964	–
f_2	1672	–
f_3	2686	–
f_4	2648	2648
f_5	2648	4440

VII. Resultados

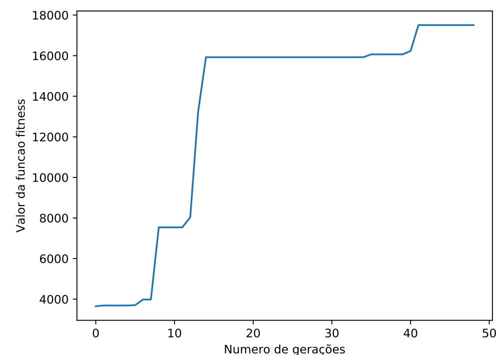
Levando em consideração o que foi discutido na Seção VI, apresentaremos nesta seção resultados mais detalhados para as funções de *fitness* f_4 e f_5, utilizando a configuração II do NEAT.

Na Figura 8 apresentamos o número de gerações versus o valor máximo da função *fitness* para as funções f_4 e f_5. Embora os valores das funções não possam ser comparados — pois não estão na mesma escala, por serem funções com pesos diferentes— podemos comparar o *comportamento* apresentado por elas. Na Figura 8a é possível ver que da geração 0 até a 10, o valor da função *fitness* cresceu rapidamente, enquanto que da geração 10 até a 45 (aproximadamente), permaneceu quase que constante (com leves crescimentos) e, entre as gerações 45 e 50 volta a crescer rapidamente, depois disso mantendo-se estável. Esse comportamento é muito similar ao da Figura 8b, embora existam algumas diferenças em intensidades de crescimento.

No entanto, quando olhamos os valores da Tabela III, percebemos que, com aproximadamente a mesma quantidade de gerações, a função f_5 foi capaz de guiar o Mega Man mais à direita do que em comparação com a f_4 e, com isso, podemos concluir que a melhor função de *fitness* avaliada foi a f_5. Analisando os *replays* dos melhores indivíduos para as duas funções, percebemos que quando utilizamos a função f_4, embora o Mega Man conseguissem matar o primeiro sub-chefe, ele não conseguia escalar a grande parede, enquanto que com a função f_5, ele conseguiu.



(a) Função *fitness* f_4



(b) Função *fitness* f_5

Figura 8: Número de gerações versus o valor máximo da função *fitness* para as funções f_4 e f_5.

De acordo com os experimentos realizados, verificamos diferentes resultados ao longo do processo evolutivo. Nos primeiros indivíduos, o Mega Man realizava movimentos básicos e morria rapidamente, não sendo capaz de andar e chegar ao primeiro sub-chefe, o que era refletido pela falta de complexidade das redes. Próximo de 48 gerações (no caso da função f_5) obtivemos os melhores indivíduos, para os quais observamos uma rede muito mais complexa com vários nós e conexões, como podemos ver na Figura 9.

Na Figura 10 apresentamos algumas capturas de tela ao longo do processo evolutivo, no qual o Mega Man aprendeu a pular buracos, (Figura 10a), matar inimigos (Figura 10b), matar ao primeiro sub chefe (Figura 10c) e escalar paredes altas (Figura 10d).

Finalmente pouco após matar o segundo sub-chefe, o Mega Man avança, mata alguns inimigos, pula alguns buracos, mas morre ao cair em um buraco, como mostrado na Figura 11.

VIII. Conclusões e Trabalhos Futuros

Neste trabalho, realizamos diversos testes com o algoritmo NEAT a fim de ensinar uma IA a terminar a primeira fase

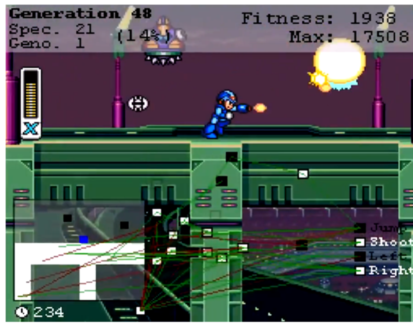
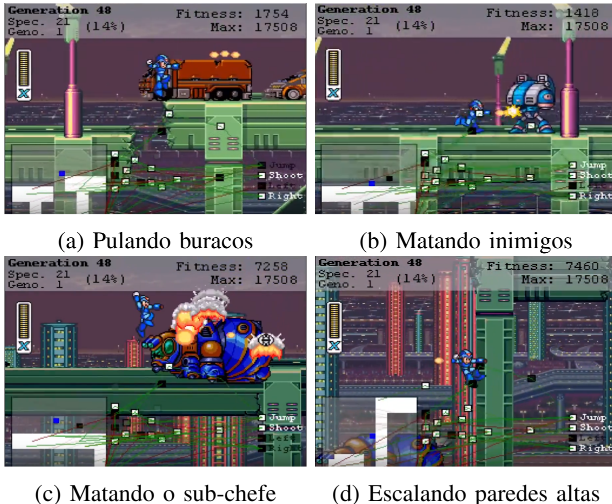


Figura 9: Arquitetura da Rede Neural do melhor indivíduo.



(a) Pulando buracos (b) Matando inimigos

(c) Matando o sub-chefe (d) Escalando paredes altas

Figura 10: Capturas de tela do processo evolutivo.

do jogo MMX. Funções mais básicas de *fitness* não foram suficientes para guiar os indivíduos a aprender mecânicas básicas do jogo, como matar inimigos ou escalar paredes. Já funções mais complexas permitiram com que o Mega Man avançasse mais na fase, conseguindo matar dois sub-chefes e escalar paredes altas, além das mecânicas básicas.

No entanto, nem os melhores indivíduos foram capazes de chegar até o final da fase. Devido ao tempo necessário para o desenvolvimento das funções de *fitness*, bem com o tempo de processamento necessário para testá-las, não conseguimos avaliar se com mais gerações para a melhor função de *fitness*, f_5, o Mega Man teria sido capaz de chegar até final da fase. Essa tarefa será deixada como trabalho futuro.

Ao longo das evoluções também notamos que, ao invés de usar uma RN *fixa* para a fase inteira por indivíduo, talvez seja mais benéfico utilizar abordagens tal como o rtNEAT, que possibilitam que a topologia da RN de um mesmo indivíduo sofra alterações em tempo real ao longo da fase. A utilização de algoritmos desse tipo também será deixada como trabalho futuro.

Figura 11: Captura de tela do momento em que o Mega Man morre no melhor indivíduo para a função f_5.

REFERÊNCIAS

[1] Lai, Matthew. 2015. Giraffe: Using Deep Reinforcement Learning to Play Chess.

[2] Loos et al. 2017. Deep Network Guided Proof Search.

[3] Deng and Yu. 2014. Deep Learning: Methods and Applications. Foundations and Trends in Signal Processing: Vol. 7: No. 3–4, pp. 197-387

[4] Deng et al. 2013. Recent Advances in Deep Learning for Speech Research at Microsoft. IEEE International Conference on Acoustics, Speech and Signal Processing.

[5] Godfrey and Gashler. 2017. A Parameterized Activation Function for Learning Fuzzy Logic Operations in Deep Neural Networks. CoRR.

[6] Roberts and Chen. 2015. Learning-based Procedural Content Generation. IEEE Transactions on Computational Intelligence and AI in Games. Vol. 7: No. 1, pp. 88-101.

[7] Stanley and Miikkulainen. 2002. Evolving Neural Networks Through Augmenting Topologies. Evolutionary Computation. Vol. 10: No. 2, pp. 99-127.

[8] Han and Han. 2016. Evolving Mario to Maximize Coin Score Using NEAT and Novelty.

[9] Foley and Kuhn. 2015. A Comparison of Genetic Algorithms Using Super Mario Bros. Major Qualifying Project (BSc). Worcester Polytechnic Institute.

[10] Nidorf et al. 2010. A Comparative Study of NEAT and XCS in Robocode". IEEE Congress on Evolutionary Computation, Barcelona, pp. 1-8.

[11] Boris and Goran. 2016. Evolving Neural Network to Play Game 2048. 24th Telecommunications Forum TELFOR. Servia, Belgrado.

[12] Andersen et al. 2002. Neuroevolution through Augmenting Topologies Applied to Evolving Neural Networks to Play Othello. Technical Report HR-02-01. Department of Computer Sciences. The University of Texas at Austin, EUA.

[13] Hausknecht et al. 2014. A Neuroevolution Approach to General Atari Game Playing. IEEE Transactions on Computational Intelligence and AI in Games.

[14] Gauci and Stanley. 2008. A Case Study on the Critical Role of Geometric Regularity in Machine Learning. Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence.

[15] Stanley et al. 2005. Real-time Neuroevolution in the NERO Video Game. IEEE Transactions on Evolutionary Computation (Special Issue on Evolutionary Computation and Games). Vol. 9: No. 6

[16] Zhi et al. 2017. Evolving Deep Neural Networks. CoRR.

[17] Whitley. 1994. A Genetic Algorithm Tutorial. Statistics and Computing.

[18] Wasserman. 1993. Advanced Methods in Neural Computing. John Wiley & Sons, Inc., New York, NY, USA, 1st Ed.

[19] Bishop. 2006. Pattern Recognition and Machine Learning. Springer-Verlag New York, Secaucus, NJ.

[20] Sutton and Barto. 1998. Reinforcement Learning: An Introduction. A Bradford Book; 1St Edition edition.

[21] Gomez and Miikkulainen. 1999. Solving non-Markovian Control Tasks with Neuroevolution. In Dean, T., editor, Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, pages 1356–1361, Morgan Kaufmann, San Francisco, California.

[22] Gruau et al. 1996. A Comparison Between Cellular Encoding and Direct Encoding for Genetic Neural Networks. In Koza, J. R. et al., editors, Genetic Programming: Proceedings of the First Annual Conference, pages 81–89, MIT Press, Cambridge, Massachusetts.

[23] Moriarty and Miikkulainen. 1997. Forming Neural Networks Through Efficient and Adaptive co-evolution. Evolutionary Computation.

[24] Whitley et al. 1993. Genetic Reinforcement Learning for Neurocontrol Problems. Machine Learning, 13:259–284.