Algoritmos genéticos aplicados ao Tetris

André Almeida
RA: 164047

Igor Torrente
RA: 169820

Lucas Cunha
RA: 172655

João Spuri
RA: 155943

I. Resumo

Nesse projeto, foi proposta e experimentada uma abordagem para a criação de um algoritmo que consiga fazer jogadas no jogo eletrônico *Tetris*, obtendo uma pontuação compatível com a esperada por humanos. Para tal, foram utilizadas técnicas de algoritmos genéticos aplicadas em redes neurais artificiais.

II. Introdução

A. Tetris

1) O jogo: Tetris é um jogo eletrônico criado em 1984 pelo matemático soviético Alexey Pazhitnov, tendo obtido grande popularidade principalmente nas plataformas *Atari ST* e no *Nintendo Entertainment System* [1]. Até hoje, já foram vendidas mais de 50 milhões de cópias mundialmente. O jogo é do gênero *puzzle*, onde o jogador precisa resolver algum tipo de quebra-cabeça.

No Tetris, o "tabuleiro" do jogo é formado por uma malha de $22x10$ quadrados (com as duas linhas do topo ocultas ao jogador), onde o jogador deve ir encaixando as peças (os "Tetraminós") que caem verticalmente no tabuleiro em uma sequência aleatória. Existem 7 tipos de Tetraminós, cada um formato distinto. O objetivo do jogador é manipular essas peças, movendo-as horizontalmente e girando-as de forma a criar uma linha horizontal no tabuleiro sem espaços vazios. Quando uma linha assim é completa, ela é destruída, as peças acima dela "caem" uma linha para baixo e o jogador pontua.

Conforme as linhas vão sendo limpas, o jogador avança entre níveis mais difíceis. A cada nível, as peças caem mais rapidamente, exigindo um tempo de resposta cada vez maior do jogador. O jogo acaba quando alguma peça, por falta de espaço no tabuleiro, fica com alguma parte posicionada fora do tabuleiro (acima da linha 22).

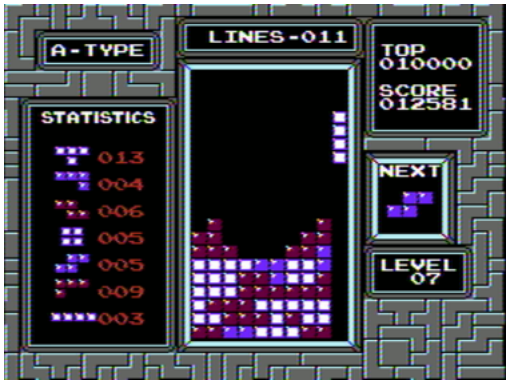

Figura 1: *captura de tela da versão* NES *do jogo*

2) Complexidade computacional: Foi provado [2] que em uma versão de Tetris onde o jogador já conhece toda a sequência de peças que virão, os seguinte objetivos são problemas NP-Completos:

- Maximizar o número de linhas limpas enquanto joga com a sequência dada;
- Maximizar o número de peças colocadas antes de completar uma linha;
- Maximizar o número de pontuações simultâneas de quatro linhas;
- Minimizar a altura da última peça colocada em uma sequência.

Com exceção do terceiro, todos esses objetivos também são difíceis de serem aproximados.

B. Algoritmos genéticos

Um algoritmo genético é uma técnica para encontrar uma solução ideal ou próxima à ideal para um problema computacional, com inspirações nas teorias darwinistas de evolução dos seres vivos. As iterações do algoritmo são sobre as gerações, onde uma geração é um conjuto de indivíduos e indivíduos são funções/modelos. Em síntese, um algoritmo genético funciona da seguinte maneira [3]:

1) Os parâmetros dos indivíduos da primeira geração são gerados randomicamente;
2) Algum tipo de função de custo é aplicada para avaliar cada indivíduo. Essa função é usada para determinar o sucesso dos indivíduos;
3) Alguma porcentagem dos melhores indivíduos (segundo seus resultados do item anterior) é escolhida.
4) Os indivíduos escolhidos no item anterior serão os "pais" da nova geração. A partir deles, combinações e mutações irão gerar os outros indivíduos da nova geração.
5) Repita os passos 2-4 até alguma condição de parada for atingida. Quando isso acontecer retorna o melhor indivíduo da última geração.

C. Redes neurais artificiais

Rede neural artificial é uma coleção de unidades chamadas neurônios que são interligados entre si com uma ordem (camadas), com inspiração biológica no funcionamento do cérebro dos animais. Cada neurônio de uma camada se liga a todos os neurônios da anterior (se não for a camada de entrada) e da posterior (se não for a camada de saída). Cada neurônio é ativado de acordo com uma função de ativação, os parâmetros dessa função de ativação são as saídas dos neurônios anteriores multiplicados por um peso nas arestas

que conectam estes neurônios. Nos neurônios de entrada são inseridos metadados e nas funções de saída algum tipo de resposta ou classificação [4].

D. Aprendizado de máquina para jogos eletrônicos

Aprendizado de máquinas aplicado em jogos eletrônicos vem sendo estudado pela academia como forma de avaliar a capacidade da máquina de executar tão bem quantas tarefas complexas, como é o caso de jogar videogames. Um estudo em destaque é o caso do AlphaGo, uma inteligência artificial que usa técnicas de aprendizado profundo e conseguiu vencer o campeão mundial do jogo Go [5]. Alguns desses estudos utilizam a técnica de algortimos genéticos para encontrar um modelo computacional adequado para o jogo, e acabaram gerando modelos que são compatíveis com o desempenho de jogadores reais, em jogos como Snake [6] e Counter-Strike [7]. Nesses jogos, parâmetros como pontuação do jogador e desempenho estratégico são usados como função de custo para treinar o algoritmo.

E. Trabalhos relacionados

Em [8], o autor cria uma rede neural para jogar Tetris, usando algoritmos genéticos para otimizar os pesos dos neurônios (substituindo o *backpropagation*). O jogador pontua ao descer peças no tabuleiro e completar linhas. A função de custo é composta por:

- número de linhas completadas;
- altura máxima, altura acumulada;
- altura relativa, buracos das linhas;
- completude do tabuleiro.

Durante o jogo, a cada nova peça, todas as jogadas possíveis são testadas e a que retornar a melhor pontuação (seguindo as heurísticas acima) é executada.

No trabalho [9], o autor utiliza uma função com as heurísticas:

- altura agregada;
- linhas completadas;
- buracos;
- diferença de alturas vizinhas.

Novamente, todas as jogadas possíveis são testadas e a que retornar o maior valor é executada. A função utilizada é a soma do produto das heurísticas com uma variável de valor inicialmente desconhecido. Os pesos de cada uma das heurísticas foi determinado usando algoritmos genéticos.

No artigo [10], as autoras utilizam uma metodologia semelhante à vista em [9], porém com as seguintes heurísticas:

- maior pilha;
- buracos;
- buracos conectados;
- linhas removidas;
- diferença de altitude;
- profundidade máxima;
- soma dos vales;
- altura do último tetraminó;
- número de células ocupadas;
- número de células ocupadas por coluna;
- soma de transições ocupadas/desocupadas horizontalmente e verticalmente.

A função de custo é dado por $\sum_{i=1}^{n} w_i|r_i(b) - d_i|^{e_i}$, onde $r_i(b)$ é o valor retornado por cada uma das heurísticas e os valores de w_i, d_i e e_i foram descobertos utilizando algoritmos genéticos.

F. Tecnologias empregadas

Para treinamento das redes neurais, foi utilizada a versão com OpenCL[11] do Torch[12], um framework de computação cientifica para LuaJIT[13]. Reutilizamos o Tetris feito em [14], devido a simplicidade (o jogo é executado em um emulador de terminal), boa documentação e porque foi escrito em Lua, facilitando a integração dos dois programas e a modificação do jogo.

G. Abordando o problema

A partir disso, decidimos explorar o uso de redes neurais artificias combinadas com algoritmos genéticos com entradas/saídas diferentes dos outros trabalhos citados para chegarmos em resultados melhores.

III. Metodologia

Utilizamos redes neurais artificiais para fazer as jogadas automaticamentes. Como em [8], substituímos a parte do *backpropagation* pelo algoritmo genético. Nosso *crossover*, pega aleatoriamente os dez melhores indivíduos de cada população para cruzamento. Além disso, casa peso dos neurônios tem 30% de chance de ter seu valor alterado pela soma de algum valor aleatório, que seria nossa mutação. Porém, diferentemente dos outros trabalhos, a saída da nossa rede é composta por 14 neurônios: qual das 10 colunas a peça deve ser posicionada e em qual das 4 rotações possíveis. Dessa forma, pretendíamos reduzir o custo computacional de simular todas as jogadas possíveis e ser uma saída mais próxima do que um jogador humano faria. A entrada da nossa rede era, inicialmente, 221 neurônios representando o tabuleiro de 22x10 linearizado, onde um neurônio ativado indicava que havia alguma peça ocupando aquela posição e um neurônio para indicar qual o *id* da peça a ser colocada. Essa entrada tinha como objetivo ser mais parecida com o que um humano recebe ao olhar para o tabuleiro.

Para comparar o desempenho do modelo com jogadores humanos, coletamos dados de 50 jogos: pedimos que 10 pessoas jogassem 5 vezes a mesma implementação de Tetris escolhida para nossos experimentos. Com isso, obtivemos que, antes de perder o jogo, um humano chega em média ao level 5, fazendo 40 linhas e um score de 70.

Tentamos várias funções de custos, iniciando como somente o número de linhas limpadas na média de dez jogos e a pontuação padrão do Tetris como critério de desempate, porém logo percebemos que essa função insentivava as redes a somente espalharem as peças mais do que realmente limparem linhas.

Dessa forma uma nova função de custo foi formulada, na qual quanto mais ”fundo” a peça estivesse no tabuleiro, mais pontos ela valia. Porém, se a peça fosse colocada a cima da metade do tabuleiro, quanto mais alta ela fosse colocada, mais ele era penalizado. Neste caso tivemos problemas com pontuações negativas, o que levava em alguns casos a escolha errada das melhores redes.

Finalmete, a função escolhida foi a qual cada peça vale mais pontos se colocada mais no fundo, porém essa pontuação é então dividida pela altura da maior torre atual no jogo, o que leva a rede a espalhar mais as suas peças, mas também incentiva ela a construir torres menores, o que por consequência a limpar mais linhas.

Ao ordenar os indivíduos para a escolha do crossover, primeiro eles eram ordenados pelo número de linhas completadas e, em cada conjunto de indivíduos que completaram a mesma linha, eram ordenados pela pontuação. Dessa forma, a função de custo deveria escolher os indivíduos que mais inserissem peças ao fundo do tabuleiro, causando assim uma maior alocação de peças na base e, consequentemente, mais linhas concluídas e mais distante do topo (onde o jogo é finalizado).

Entretanto para a versão final da função de custo, passamos a ordenar as redes somente pela pontuação, pois percebemos que a ordenação por linhas causava a algumas boas redes que fossem eliminadas por má sorte.

Modificamos o código-fonte do jogo de forma que ele posicionasse as peças na coluna indicada e com a rotação indicada. A cada peça colocada, a rede recebia a entrada do tabuleiro e retornava uma nova configuração de coluna/rotação. Cada rede jogava o jogo até perder 10 vezes, e cada geração continha 10 indivíduos. Ao final dos 100 jogos, o crossover era utilizado para gerar a próxima geração.

A partir disto, fizemos experimentos e conforme os resultados, fazíamos as modificações que julgávamos necessárias para melhorar a precisão de nossos modelos.

IV. Experimentos e Resultados

No nosso primeiro teste, utilizamos o jogo de Tetris com todas as peças do jogo e uma variedade de configurações de rede neural (mudando o número de camadas, neurônios por camadas e função de ativação). Nenhuma rede, mesmo após 200 gerações evolutivas, conseguia completar mais de uma linha por jogo nem demonstrar algum avanço.

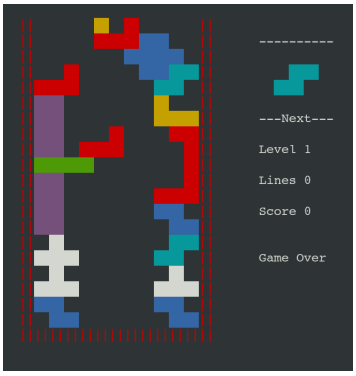

Figura 2: *Amostra da rede jogando Tetris nos primeiros experimentos*

Para testar a viabilidade do modelo, de forma a verificarmos de o modelo funcionaria em outras condições, mudamos o jogo para um modelo mais simples, com apenas peças de 1x1 e um tabuleiro de 10x10.

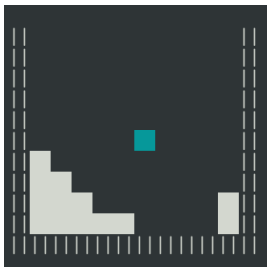

Figura 3: *Primeiro Tetris simplificado utilizado nos experimentos*

A melhor configuração de rede neural obtida nesse cenário foi uma rede com três camadas. A camada de entrada original foi modificada para uma com 22 neurônios:

- 1 com um identificador da peça;
- 10 com a altura de cada coluna;
- 9 com a diferença de altura entre cada par de colunas adjacentes;
- 1 com a maior diferença entre alturas.

A camada oculta contem 100 neurônios. Mantivemos apenas uma camada, pois a adição de mais camadas ocultas causou piora nos resultados do jogo. A saída da rede, com 14 neurônios, indica qual das 10 colunas a peça deve ser colocada e qual a rotação dela. Depois de 12 gerações, a rede conseguiu jogar o jogo ”infinitamente”. Ele completava linhas sequencialmente, da mesma forma, sem fim. Um fato curioso nessa etapa do treinamento foi que a rede se aproveitou de um *bug* que permitia completar linhas com muitos menos movimentos, que não era conhecido pelos desenvolvedores. Após corrigir o *bug*, a rede foi retreinada e o jogo obteve o mesmo resultado de jogar infinitamente na geração 39.

A partir deste cenário, utilizamos *transfer learning*. Esse conceito é utilizado em redes neurais, como forma de reaproveitar alguma parte de uma neural já treinada para outro problema parecido. Mudamos o jogo para uma outra peça, de 2x1.

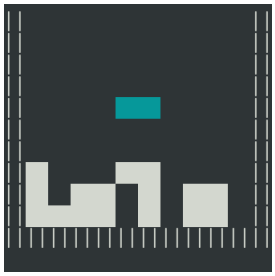

Figura 4: *Segundo Tetris simplificado*

Utilizando a mesma rede que conseguia ganhar no Tetris anterior, testamos nesse novo modelo. O *transfer learning*

funcionou e na próxima geração a rede já conseguia completar as linhas sem fim.

Contudo, a rede novamente não conseguia jogar com modelos mais complexos a partir da rede treinada nos modelos anteriores. Ao inserirmos ambas as formas, 1x1 e 2x1, a rede não conseguia avançar e novamente limpava apenas uma linha por jogo na média. Para tentar contorna esse problema, fizemos novamente um teste com apenas uma peça, mas, dessa vez, uma mais complexa. Essa nova peça é uma ”combinação” das duas peças anteriores, em um formato de *L*. Contudo, a rede tentava aplicar as mesmas técnicas de antes, sem explorar as rotações ou fazendo-as de forma sem sentido. Em um jogo que precisava mais do que apenas lógica para o posicionamento, mas também lógica para rotação, a rede não obteve sucesso.

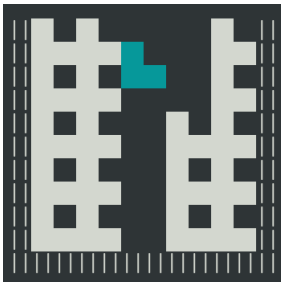

Figura 5: *Rede jogando o quarto Tetris simplificado proposto*

Ao analisarmos a saída da rede, constatamos que as redes que eram bem sucedidas nas versões simplificadas faziam apenas dois movimentos, usando apenas duas das dez posições disponíveis. A rede jogava todas as peças para a coluna mais à esquerda, depois para a mais a direita e, quando ambos lados estiverem concluídos, a rede posiciona uma peça na coluna central, limpando uma linha. Isso é possível porque, no jogo, existe um sistema de colisão. Quando uma peça tenta atingir uma posição já ocupada do tabuleiro, ela não consegue avançar lateralmente. Dessa forma, a rede consegue completar duas metades do tabuleiro apenas ordenando que as peças vão para a posição mais remota possível.

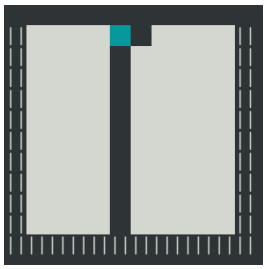

Figura 6: *Rede prestes a limpar uma linha seguindo seu algoritmo*

V. Conclusão

Segundo os resultados dos nossos experimentos, a rede falhou em aprender a jogar Tetris. Apesar de conseguir jogar jogos simplificados (com apenas uma peça), ela não posiciona as peças conforme as informações retirados do tabuleiro, apenas segue uma ordem de passos que irá obter sucesso, sem depender de dados do tabuleiro. Jogos com mais de uma peça não são determinísticos, já que as peças vem de forma aleatória. Nesse cenário, um algoritmo como o executado pela rede não funciona, já que peças diferentes requerem respostas diferentes. Já em um jogo com apenas uma peça, todos os passos de sucesso são determinísticos, já que a mesma peça virá sempre. Devido as muitas variações que os jogos não determinísticos oferecem, durante o treinamento dos indivíduos da população (as redes), eles, na prática, nunca encontravam o mesmo jogo, o que pode ser um fator que dificultava o aprendizado. A abordagem de simular todas as jogadas possíveis e fazer a que tem o melhor desempenho segundo a função de custo provavelmente contorna este problema, apesar de ser contrário a nossa proposta de resolução do problema.

A métrica utilizada para determinar o sucesso de nosso modelo (comparar com resultados humanos) não fez muito sentido na prática, pois, pelos resultados obtidos, uma rede que aprenda a jogar jogo bem provavelmente não irá cometer erros e irá jogar infinitamente, até atingir a pontuação máxima permitida no jogo.

VI. Estudos futuros

Para melhorar os resultados de nossa abordagem, serão necessários novos estudos em relação a nossa rede. Uma das alternativas a ser testada é o uso de redes convulacionais [15]. Esse tipo de rede é utilizado principalmente com problemas relacionados à imagens, porque ela mantem informações importantes da localização entre pixels. Dessa forma, a rede não recebe apenas a matriz de pixels linearizada, mas consegue criar uma relação entre os pixels vizinhos. Isso é importante para o tabuleiro do jogo, já que pontos com/sem peça tem mais sentido sabendo as informações das casas vizinhas.

Também devem ser testados novas escolhas de funções de custo, já que em nosso trabalho focamos na função *sigmoid*.

Outra alternativa para a melhora da rede neural seria usar a o algoritmo *NeuroEvolution of Augmenting Topologies* (NEAT)[16], que além de alterar os pesos dos neurônios da rede neural, ela ”evolui” a rede, adicionando ou removendo neurônios seja de uma camada já existente, seja em uma nova, com ligações entre os neurônios. Essa rede usa um *crossover* mais complexo para a reprodução das redes, o que torna sua implementação mais complicada e mais lenta, mas ela tem grande potencial de chegar em uma rede complexa e adaptada o suficiente para nosso problema.

VII. Código e dados

Todo o código do experimento pode ser encontrado em: `https://github.com/andrealmeid/AutoTetris`

References

[1] `http://www.atarihq.com/tsr/special/tetrishist.html`

[2] Demaine, E. D., Hohenberger, S., & Liben-Nowell, D. (2003, July). Tetris is hard, even to approximate. In COCOON (pp. 351-363).

[3] Goldberg, D. E. (1989). Genetic Algorithms in Search, Optimization, and Machine Learning.

[4] Geron, A. (2017). *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems.*

[5] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., ... & Chen, Y. (2017). Mastering the game of go without human knowledge. Nature, 550(7676), 354-359.
[6] Yeh, J. F., Su, P. H., Huang, S. H., & Chiang, T. C. (2016, November). Snake game AI: Movement rating functions and evolutionary algorithm-based optimization. In Technologies and Applications of Artificial Intelligence (TAAI), 2016 Conference on (pp. 256-261). IEEE.
[7] Cole, N., Louis, S. J., & Miles, C. (2004, June). Using a genetic algorithm to tune first-person shooter bots. In Evolutionary Computation, 2004. CEC2004. Congress on (Vol. 1, pp. 139-145). IEEE.
[8] https://github.com/llSourcell/How_to_make_an_evolutionary_tetris_bot
[9] https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player/
[10] Böhm, N., Kókai, G., & Mandl, S. (2005). An evolutionary approach to tetris. In The Sixth Metaheuristics International Conference (MIC2005) (p. 5).
[11] https://www.khronos.org/opencl/
[12] http://torch.ch/
[13] http://luajit.org/luajit.html
[14] https://github.com/tylerneylon/termtris
[15] LeCun, Y. (2015). LeNet-5, convolutional neural networks. URL: http://yann. lecun. com/exdb/lenet.
[16] Stanley, K. O., & Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. Evolutionary computation, 10(2), 99-127.