PROJETO FINAL - IA

Relatório

Trabalho realizado por:
Ana Toscano - a22207260
Francisco Pinto - a22207132
Maria Inês Silva - a22207550

Junho 2024

Índice

1 Divisão do Trabalho

* Ana Toscano e Maria Inês Silva:

-Pesquisa, Implementação e Desenvolvimento do Código;

-Realização do relatório;

-Discussão de resultados.

* Francisco Pinto:

-Realização do relatório;

-Discussão de resultados.

2 Introdução

Este relatório descreve um projeto de desenvolvimento de um jogo representado numa grelha 5x5, onde cada célula tem uma probabilidade de 30 por cento de conter lixo. O objetivo do jogador, ou do agente de Inteligência Artificial (IA), é recolher o maior número possível de lixos num máximo de 30 passos por jogo.

O jogo começa com o Iron, posicionado aleatoriamente na grelha que pode ser movido em quatro direções: cima, baixo, esquerda e direita. O desafio proposto consiste em otimizar os movimentos para maximizar a apanha de lixo dentro do limite de passos.

Existem dois modos de jogo: com um jogador humano ou com um agente de IA. No modo de IA, o agente aprende a partir de jogos anteriores utilizando redes neuronais e algoritmos genéticos. Durante o processo de aprendizagem, os pesos e bias das redes neuronais são ajustados para melhorar a performance do agente ao longo das várias gerações de treino.

O desempenho do agente de IA é avaliado com base na pontuação média de aptidão (fitness score) e na quantidade de lixo não apanhado após cada geração. Para observar a performance do agente de IA, os resultados podem ser visualizados através de um gráfico que mostra a evolução da aptidão e da eficiência na apanha de lixo ao longo das gerações.

3 Metodologia

Para ser possível a implementação do jogo decidimos dividir o código em 5 ficheiros diferentes, vamos então falar sobre todos eles.

3.1 RedeNeuronal.py

Para começar vamos explicar o que faz a rede neuronal. Uma rede neuronal é um modelo feito para reconhecer padrões e tomar decisões com base nos dados de entrada. Esta é composta por várias unidades de processamento simples (neurónios) organizadas por camadas relacionadas. Os neurónios recebem um conjunto de entradas ponderadas, aplicam uma função de ativação nessa entrada(input) e produzem uma saída (output).

Já as camadas têm três tipos principais:

- Entrada(input) – recebe os dados e passa para a seguinte
- Oculta (hidden layer) – camada intermediária entre a entrada e saída e onde ocorre o processamento dos dados.
- Saída(output) – fornece a saída final da rede apos os processamentos nas camadas ocultas.

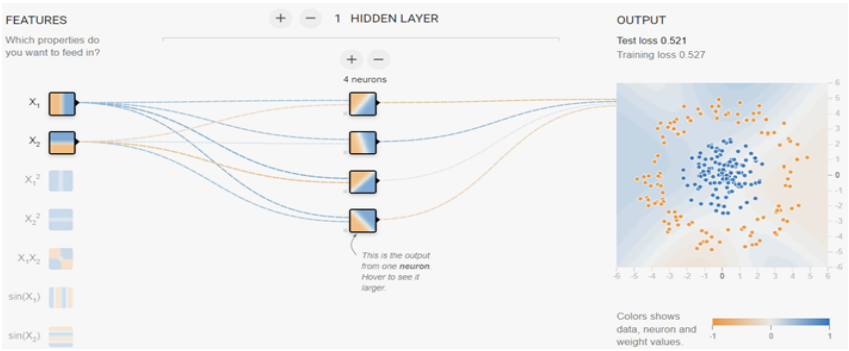

Figura 1: Demostração da rede neuronal

Para cada conexão de neurónios há um peso associado(weight) que determina a importância da entrada para a saída do neurónio. Estes pesos são ajustados durante o treino da rede de modo que o desempenho do modelo seja otimizado.

Para alem dos pesos é ainda adicionado a cada neurónio (tirando na camada de entrada) um parâmetro que permite que a rede aprenda a função de ativação pretendida, a este parâmetro damos o nome de bias.

Para a criação da rede neuronal é ainda necessário o processo de feedfoward. Este trata-se da propagação dos dados de entrada através da rede até a camada de saída onde é gerada uma previsão ou classificação.

O nosso código define então uma rede neuronal que implementa métodos de modo que seja possível realizar operações de feedfoward e obter os pesos e bias da rede. A nossa rede possui já uma camada de entrada de tamanho 4 (vizinhos do iron), camada oculta de 6 e camada de saída 4 (movimentos possíveis).

Para começar implementamos as funções de ativação relu e softmax:

- A função relu(a) é a que vai retornar o máximo entre 0 e cada elemento de 'a'.
- A função softmax(a) serve para normalizar um array 'a' em probabilidades.

De seguida inicializamos a classe através do método init(self, sizes). Este método recebe uma lista de tamanhos que vai ser o que vai especificar o número de neurónios em cada camada. Este inicializa os pesos(self.weights) e os bias(self.bias) para cada camada com valores aleatórios através da utilização do np.random.randn.

Depois da inicialização passamos para o método feedfoward em que:

Camada de entrada - os dados de entrada (inputs) são convertidos num vetor coluna(a)

Camada oculta - para a camada oculta calculamos o produto entre os pesos (w) e a entrada (a) adicionando o bias (b) e aplicamos ainda a função de ativação relu:

- Z = w * a + b
- a = relu(z)

este passo é repetido para cada camada oculta guardando os valores de ativaçao na lista 'hidden values'.

Camada de saida - nesta camada calculamos novamente o produto entre os pesos (self.weights[-1]) e a ativação da ultima camada oculta (a), adicionamos ainda o bias (self.bias[-1]) correspondente:

- Output = w * a +b

De seguida aplicamos a função softmax para obter as probabilidades normalizadas para cada classe:

- Outputs = softmax(output)

Encontramos o índice do neurónio de saida com maior probabilidade através da função argmax(outputs) de modo que possamos saber o movimento previsto (y hat).

Por fim, ainda nesta função, retornamos várias informações como: o índice de movimento previsto, o movimento como string, as saídas da softmax, os valores de ativação das camadas ocultas e o índice do neurónio de saída com maior probabilidade.

3.1.1 Fluxograma

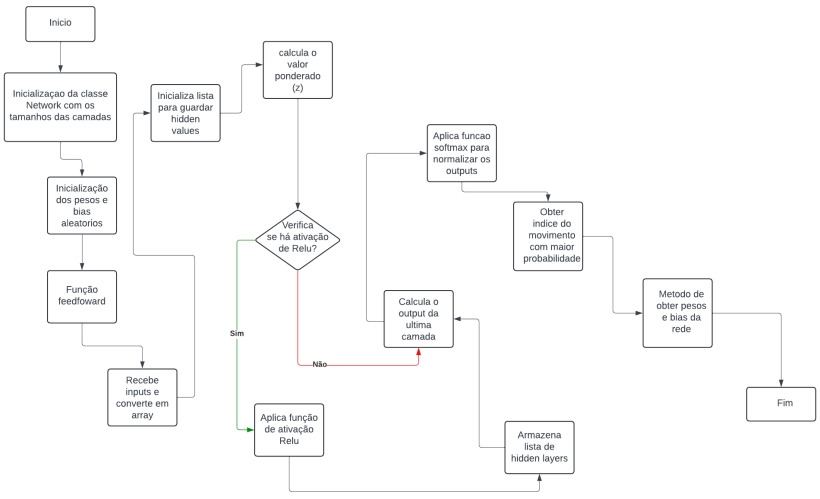

Figura 2: Fluxograma da nossa rede neuronal

3.2 genetic algorithm.py

Algoritmos genéticos são técnicas de busca e otimização. É a busca da melhor solução para um certo problema e consiste em tentar várias soluções e utilizar a informação obtida para conseguir melhores soluções.

Estas técnicas de otimização mostram:

- Espaço de busca – onde estão todas as possíveis soluções do problema.
- Função de objetivo – utilizada para avaliar as soluções obtidas associando a cada uma delas uma nota.

Este algoritmo trabalha com uma população de soluções simultaneamente e utiliza apenas as informações de custo e recompensa. Ou seja, ele funciona com uma população inicial de soluções possíveis (genes), onde cada solução tem um conjunto de parâmetros (DNA) que pode ser ajustado consoante as nossas necessidades de maneira a melhorar o desempenho ao longo das diversas gerações.

- População inicial – esta é composta por um número fixo de genes cada um representando uma solução possível para o problema.

  Esta população é formada através da função 'create initial genes()', esta cria uma lista de 'genes' com 'num of genes' pedidos da classe 'network' e inicializados com tamanhos de camada específicos.

- Seleção – Neste passo, durante cada geração, os genes com melhor desempenho (fitness score) são selecionados para se reproduzir.

- Crossover – este serve para combinar as informações genéticas de dois pais para criar descendentes. Nesta função ('crossover'), primeiro recebe dois genes pais ('parent1' e 'parent2'), obtém os seus DNAS ('dna1' e 'dna2') e de seguida realiza o crossover num ponto aleatório criando um dna que é utilizado para inicializar um novo gene filho.

- Mutação – esta vai ser a que vai introduzir pequenas alterações aleatórias nos dnas dos genes para explorar novas possibilidades no espaço de soluções. Nesta função ('mutate') cada gene tem uma pequena chance de sofrer uma mutação em cada geração. Ou seja, tem uma probabilidade dada por 'MUTATION RATE' de ser aumentado por um pequeno valor aleatório.

- Fitness score – é a medida que avalia o quão bem cada gene se esta a sair. Quanto melhor fitness score tiverem os genes maior vai ser a probabilidade de serem selecionados na próxima geração.

Nesta função ('generate next generation') os melhores genes da geração atual são selecionados, tendo em conta o fitness score, realiza o crossover e a mutação para criar os genes na próxima geração até o número de genes ('NUM OF GENES') pretendido ser atingido.

3.2.1 Fluxograma

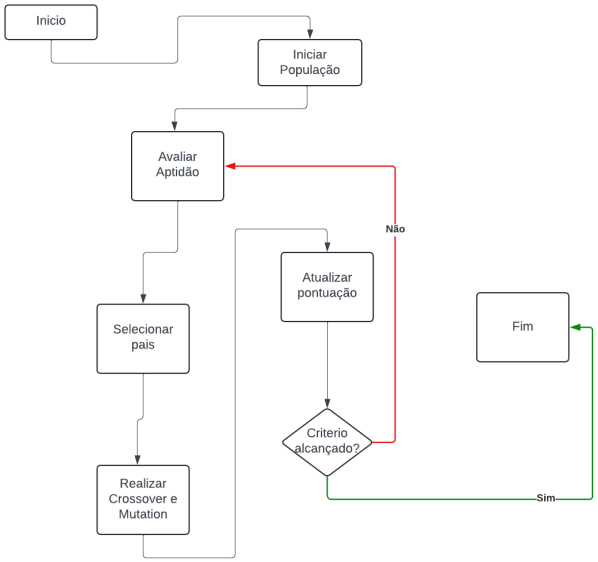

Figura 3: Fluxograma do nosso algoritmo genético

3.3 Gene.py

Nesta classe guardamos um objeto de rede neuronal e dá-nos métodos específicos para interagir dentro de um algoritmo genético, permitindo a evolução e otimização automática das soluções.

Inicio – cada instância da classe é começada com uma configuração única de rede neuronal

Avaliação – através do método 'evaluate fitness' determinamos o quão adequado um gene é em relação ao problema a ser otimizado. Este ajusta a pontuação de fitness tendo em conta as interações do gene no ambiente.

Manipulação do dna – através dos métodos 'get dna' e 'set dna' é nos possível aceder e alterar os parâmetros da rede neuronal associada a cada gene, o que facilita na evolução e adaptação das soluções ao longo das gerações.

Focando mais na nossa classe, implementámos diversas funções tais como:

- 'get dna' - transforma pesos e bias da rede neuronal numa lista unidimensional ('dna') de modo a representar o conjunto de parâmetros da rede.
- 'set dna' - atualiza os pesos e os bias da rede com base numa lista de dna fornecida configurando os parâmetros conforme os tamanhos das camadas.
- 'get network' - retorna o objeto da rede neuronal associada ao gene.
- 'set left garbage' e 'get left garbage' - métodos para aceder e modificar o atributo do 'left garbage' que pode guardar informações adicionais sobre o ambiente.

- 'reset fitness score' - volta a iniciar a pontuação de fitness a 0.
- 'get life generation' e 'increment life generation' - métodos para aceder e aumentar o contador de gerações vividas pelo gene.
- 'set dna position' - permite alterar um valor específico no dna
- 'catch garbage' - aumenta a pontuação em 5 pontos quando realiza uma ação positiva.
- 'bad move' - penaliza a pontuação em 1 ponto sempre que realiza um movimento mau.
- 'move' - usa a rede neuronal para prever o movimento tendo em conta os inputs fornecidos.
- 'evaluate fitness' - volta a iniciar a pontuação de fitness e aumenta a pontuação com base nas ações positivas e diminui tendo em conta as ações negativas.

3.4 Matrix.py

Esta vai ser a classe que vai tornar possível a representação do ambiente do jogo. Composta por células que podem ter diferentes elementos, como o Iron, o lixo e espaços vazios.

Começamos por inicializar a matriz através do método init, definindo as dimensões e o estado inicial da mesma, os parâmetros incluem a altura (' height' armazena a altura da matriz), largura (' weight' armazena a largura da matriz) e o tamanho dos ícones (' iconSize' armazena o tamanho dos ícones), já as variaveis ' matrizRows' e ' matrixCols' são quem vais calcular o numero de linhas e colunas na matriz dividindo a largura e a altura pelo tamanho do ícone, por fim as variáveis 'yIron' e 'xIron' guardam as coordenadas do 'Iron' e o ' left garbage' conta quando lixos faltam.

A função 'get left garbage' fornece-nos o número de lixos que faltam e a função 'decremente left garbage' diminui o número de lixos restantes.

A função 'fillEmptyMatrix' preenche a posição com 'empty' enquanto a função 'fillIronMatrix' posiciona o 'Iron' na matrix, sendo que se o 'Iron' já estiver na matriz a posição anterior vai ser preenchida com 'empty' e caso a nova posição seja fora dos limites da matriz vai ser posicionado na borda oposta.

A função 'fillGarbageMatrix' posiciona o lixo 'garbage' na matrix e a função 'spwanGarbage' posiciona vários lixos nas posições indicadas na matriz.

A função 'drawMatrix' é a que vai desenhar a matriz colocando as imagens nas posições de 'Iron' e 'garbage' desenhando as linhas da matriz. Esta recebe como parâmetros a imagem do 'Iron' (' image iron man'), a imagem do lixo (' image garbage') e o tamanho dos ícones ('iconSize').

Por último, a função 'get neighboors' retorna-nos a lista dos vizinhos do 'Iron'. Esta vai conter valores de:

- 0 – 'empty'
- 1 – 'iron'

- 2 – ‘garbage’
- -1 – posições fora dos limites

3.5 Run.py

Este ficheiro é o que utiliza o algoritmo genético e a rede neuronal implementados nos ficheiros (’redeneuronal’ e ‘genetic algorithm’) de modo a otimizar o ‘Iron’ no ambiente simulado. O ‘Iron’ deve mover-se na matriz para apanhar o lixo ‘garbage’.

O código acaba por estar dividido por:

- Definição de parâmetros - definimos diversos parâmetros para o algoritmo genético e a rede neuronal.
- Inicialização de genes - os genes iniciais são criados e as listas para guardar as pontuações medias de aptidão e quantidade de lixo restante são inicializadas.
- Formato do pygame - o pygame acaba por ser configurados com um tamanho específico e ícones para o ‘Iron’ e o ‘garbage’ são carregados e redimensionados. A fonte é também inicializada para mostrar o texto na tela.
- Gerações - este loop é iniciado para cada geração. Para cada gene, um novo objeto da matriz é criado, o ‘Iron’ é posicionado na matriz, e o lixo distribuído tendo em conta as posições já especificadas. Depois o ‘Iron’ realiza movimentos até todos os lixos serem apanhados ou até chegar ao número máximo de passos.
- Os inputs para a rede do gene são obtidos através da função ‘get neighboors’ (implementado em Matrix), e o movimento do gene é então escolhido. Tendo em conta o movimento a nova posição vai ser atualizada e caso um lixo for encontrado a pontuação aumenta.
- Desenho na matriz - quando a matrix é desenhada, as informações acerca da geração, pontuação e quantidade de lixo são também mostradas.
- Seleção, mutação e reprodução - Depois de todos os genes serem testados, os genes com uma melhor classificação vão ser os selecionados para a próxima geração fazendo assim com que as próximas gerações tendam a ser cada vez melhores.

Por fim, guardamos os resultados num ficheiro JSON.

Neste ficheiro incluímos o dna da última geração, as listas de pontuações medias de aptidão e as de quantidade de lixo que falta apanhar.

Para a visualização da pontuação media de aptidão ao longo das gerações para que seja possível identificar onde este obteve a melhor aptidão.

3.6 Diagrama UML

Este diagrama mostra-nos as classes principais e as suas relações no jogo.

A classe Matrix é a responsável pelo ambiente do jogo. A classe Network implementa a rede neuronal usada pelo 'Iron' de modo a ser possível tomar decisões. A classe gene inicializa um objeto na rede neuronal juntamente com outros métodos de maneira a ser possível calcular a aptidão e aplicar processos genéticos. Adicionamos ainda a função 'genetic algorithm' pois esta esta interligada a maior parte das classes.

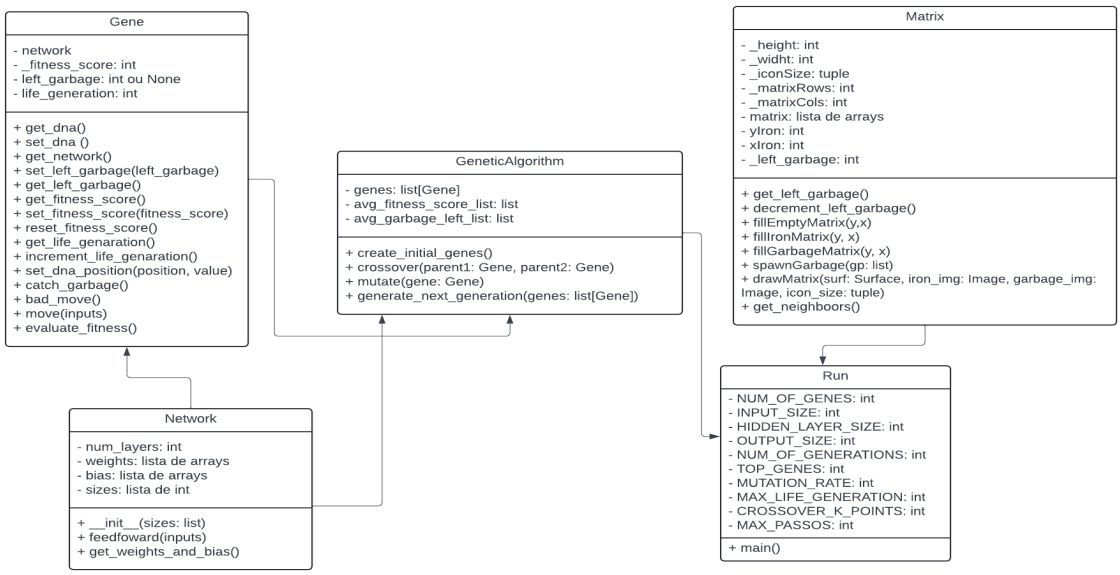

Figura 4: Diagrama UML do código

4 Resultados e Discussão

Nesta secção, apresentamos e discutimos os resultados obtidos durante as simulações com o agente de IA, a análise concentra-se na pontuação média de aptidão ao longo das gerações e na capacidade do agente de IA de imitar e potencialmente superar o desempenho humano.

4.1 Desempenho do Agente de IA

Os resultados da Figura 5 mostram a evolução da pontuação média de aptidão do agente de IA ao longo de 20 gerações. A pontuação média de aptidão, que representa a quantidade de lixo apanhado pelo agente em cada jogo, apresenta variações significativas entre gerações, mas uma tendência geral de melhoria é observada ao longo do tempo.

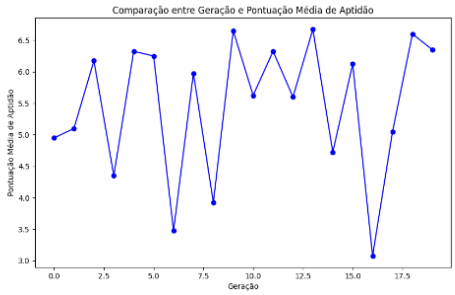

Figura 5: Comparação entre Geração e Pontuação Média de Aptidão

4.2 Análise dos Resultados

O gráfico indica que a pontuação média de aptidão varia entre 3.0 e 6.5 ao longo das gerações. Essa variabilidade pode ser atribuída ao processo dos algoritmos genéticos, que inclui elementos de seleção, cruzamento e mutação. Apesar dessas variações, há uma clara tendência de aumento na pontuação média de aptidão, sugerindo que o agente está a aprender e a melhorar a estratégia de apanha de lixo.

Os resultados foram obtidos através da simulação de várias gerações.

- Número de Genes: 200
- Número de Gerações: 20
- Top Genes: 80
- Taxa de Mutação: 0.05
- Crossover k pontos: 1
- Máximo de Passos por Simulação: 30

4.2.1 Evolução do Fitness Score e Lixo Restante

O gráfico mostra a evolução do fitness score médio e a quantidade média de lixo restante ao longo das gerações:

- Fitness Score Médio: Aumento gradual ao longo das gerações.
- Lixo Restante Médio: Diminuição ao longo das gerações.

4.3 Discussão

Os resultados indicam que o algoritmo genético foi eficaz em melhorar a eficiência do Iron na apanha de lixo, embora existam vezes que ao correr o código aparece uma linha constante o que mostra que não ha variação. O aumento do fitness score e a diminuição do lixo restante são indicadores positivos. Algumas observações inesperadas incluíram variações significativas de desempenho entre diferentes execuções devido à aleatoriedade introduzida pelo processo de mutação.

5 Conclusões

Este projeto demonstrou a eficácia do uso dos algoritmos genéticos para aprimorar a ação do robô na recolha dos lixos. A combinação de redes neuronais e algoritmos genéticos permitiu a evolução de estratégias eficientes na recolha do lixo ao longo de várias gerações. O aumento do fitness score e a redução do lixo restante indicam uma melhoria significativa na eficácia do robô.

Os resultados mostraram-nos que o agente IA foi melhorando a sua aptidão e desempenho ao longo dos vários treinos. No entanto, apesar de irem melhorando não chegámos à aptidão desejada pois, o agente IA também ainda dá vários passos desnecessários o que indica que o modelo foi progredindo, mas poderia ainda ser melhor.

Concluindo, conseguimos desenvolver/resolver o problema pedido e embora não tenhamos chegado ao modelo perfeito, este foi mostrando um melhor desempenho.

Este trabalho ajudou-nos a perceber melhor então os algoritmos genéticos e as redes neuronais e a perceber exatamente o que cada um faz.

6 Agradecimentos

Ao longo do trabalho, sempre que precisámos de ajuda, a mesma nunca foi negada e por isso gostaríamos de agradecer a todos que nos apoiaram durante a realização deste trabalho. Em especial, gostaríamos de agradecer aos nossos professores, pelo conhecimento compartilhado, orientação e apoio contínuo ao longo do projeto, foram fundamentais para garantir qualidade deste trabalho e também agradecer aos nossos colegas que, direta ou indiretamente, contribuíram para a concretização deste projeto.

7 Referências

Para o desenvolvimento deste trabalho, contámos significativamente com o suporte do ChatGPT (ChatGpt - https://chat.openai.com/), facilitou o esclarecimento de dúvidas e a realização de pesquisas e a obtenção de informações adicionais, complementando o conhecimento adquirido em sala de aula.

Para a criação dos fluxogramas e do diagrama UML utilizamos o site LucidChart (https://www.lucidchart.com)

Por fim, utilizamos tambem os slides que nos foram fornecidos nas aulas.