Carro autônomo utlizando NEAT

Bruno Libardoni
Ciência da Computação
Instituto Federal Catarinense
Videira, Santa Catarina
Email: brunolibardoni16@gmail.com

***Resumo*—Múltiplos algoritmos para gerar comportamentos de agentes de jogo foram produzidos nesses últimos anos. Grande parte deles se baseiam em Inteligência Artificial que precisam de treinamento. Neste contexto, este proposto trabalho apresenta uma estratégia de treinamento para desenvolver carros autônomos utlizando o algoritmo revolucionário NEAT capaz de completar o circuito sem que o carro se locomova para fora da pista. Com o algoritmo NEAT foi usado para encontrar a arquitetura de rede neural mais simples que consiga jogar o jogo impecavelmente. O algoritmo teve um tempo de convergência baixa atingindo o comportamento perfeito no jogo.**

***Palavra-chave*—Algoritmo; Rede; Inteligência Artificial; Arquitetura;**

I. Introduction

Python é a linguagem na vanguarda de pesquisa da AI, onde será possível encontrar a maioria das estruturas de machine learning e deep learning [1]. Deste modo, é possível usufruir de bibliotecas, métodos ou função prontas que auxiliam no momento do desenvolvimento. Com o auxílio de ferramentas já desenvolvidas e aplicadas em rede neural o proposto projeto final será desenvolvido na linguagem de programação Python.

O presente projeto final utiliza um algoritimo que faz a criação das redes neurais artificiais chamado de NEAT, juntamente com o pygame e outras bibliotecas padrões que foram necessárias na hora da implementação do trabalho. O projeto é um carro autônomo utilizando uma inteligência artificial que por meio de gerações faz a apredizagem do circuito de corrida, como dirigir e que por meio deste aprendizado a IA consiga completar o circuito de maneira que o carro não desvie da rota e da pista.

Neste trabalho, é aplicado um algoritmo de Neuroevolução conhecido como Neuroevolução de Topologias Aumentadas do acrônimo NEAT no ambiente do jogo de corrida. Com este algoritmo, além de encontrar um agente simples para jogar o jogo, encontra-o rapidamente, ou seja, poucos gerações e jogando cenários curtos. NEAT é um algoritmo originário da Rede Neural Artificial (RNA), mas possui um otimizador de arquitetura dentro dele. O otimizador é feito de Algoritmo Genético (GA) e é necessário para encontrar a arquitetura mais eficaz para RNA, uma arquitetura que tem baixo custo em termos de tempo de cálculo e complexidade [2] .

Na sessão II, será analisado trabalhos relacionados que usam neuroevolução (NEAT) em outros jogos. Na sessão da metodologia será detalhada o propósito da estruturação do jogo e da configuração do NEAT. Nos resultados será apresentados as gerações, fitness e o tempo levado que o carro autônomo levou até a colisão.

II. Trabalhos relacionados

A aplicação da Neuroevolução em jogos vem sendo utilizada há algum tempo com resultados satisfatórios na criação de agentes inteligentes capazes de jogar em nível humano e até super-humano [3]. Assim, NEAT é uma das técnicas mais promissoras no âmbido de jogos.

Flappy Bird é um jogo popular originalmente designado para dispositivos móveis. Hoje, o jogo já foi implementado em diversos tipos de linguagem de programação e tornou-se um meio ambiente de diferentes campos de teste com a inteligência artificial. A literatura [5] contém referencia de treinamento mínimo do jogo Flappy Bird indefinidamente com NEAT. No trabalho, propôs um desenvolvimento de um player virtual autônomo usando NEAT algoritmo neuroevolicionário capaz de jogar o jogo Flappy Bird. Desse modo, foi utilizado NEAT para encontrar a arquitetura de rede mais simples que consiga jogar perfeitamente o jogo. A modelagem do cenário a função de fitness foram definidas para garantir uma representação adequada do problema em comparação com o jogo real.

NEAT também pode ser aplicado ao paradigma multiobjetivo em certos jogos, onde NPCs (personagens não jogáveis) devem realizar mais de uma tarefa, como Ms. Pac-Man [7]. Neste jogo, é utilizada uma variação conhecida como Modular Multiobjetivo NEAT em que os tipos de busca sempre buscarão um agente com comportamento multimodal [8].

Um outro gênero da aplicação de neuroevolução é em jogos de lutas. Na literatura [2], é usado NEAT para resolver o problema do algoritmo KNN chamado MizunoAI da incapacidade de ajustar seus parâmetros automaticamente. Na implementação, NEAT recebe 6 entradas de distância no eixo X e Y, assim a saída sendo a probabilidade de todos os movimentos que o oponente seja capaz de realizar. Deste modo, todas as saídas serão processadas pelo simulador para determinar a melhor contramedida baseadas nas predições.

III. Metodologia

Nesta sessão será apresentada a metodologia das fases na hora da implementação da IA, do circuito e do carro. Na primeira fase no desenvolvimento do projeto final, foi estabelecer quais ferramentas usar na implementação. Deste modo, foi escolhido NEAT pela simplicidade da estrutura.

NEAT utiliza especiação para evitar convergência prematura para soluções subótimas. As inovações potenciais são protegidas, dando às estruturas mais novas uma oportunidade melhor de se desenvolver, em vez de serem descartadas no início em favor de estruturas existentes mais desenvolvidas. Isso é feito permitindo que os indivíduos competam principalmente com outros membros de sua espécie, em vez de com a população inteira. O número de descendentes permitido por espécie é proporcional à média aptidão dessa espécie [9].

Para o uso do algoritmo NEAT é preciso de apenas dois arquivos, como mostrado na Fig 1. Na main encontra-se o desenvolvimento do código fonte e o arquivo de configuração é representada como, uma linguagem de análise de arquivo de configuração básica que fornece uma estrutura semelhante à que você encontraria em arquivos INI do Microsoft Windows [5]. Para fazer o uso do mesmo, é possível obter através da documentação ou pelo download do próprio site do NEAT.

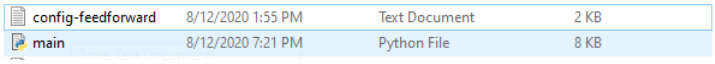

Fig. 1. Estrutura para utilização do NEAT.

No arquivo de configuração do NEAT, como mostrado na Fig 2 e Fig 3, tem-se diversos parâmetros que na própria documentação é explicado a função de cada um deles. Estes parâmetros deverão ser estabelecidos para o funcionamento da IA e na ordem de como é proporcionada na documentação. É possível fazer modificações dos parâmetros de acordo com sua necessidade.

```
fitness_criterion     = max
fitness_threshold     = 100000
pop_size              = 30
reset_on_extinction   = True

[DefaultGenome]
# node activation options
activation_default      = tanh
activation_mutate_rate  = 0.01
activation_options      = tanh

# node aggregation options
aggregation_default     = sum
aggregation_mutate_rate = 0.01
aggregation_options     = sum

# node bias options
bias_init_mean          = 0.0
bias_init_stdev         = 1.0
bias_max_value          = 30.0
bias_min_value          = -30.0
bias_mutate_power       = 0.5
bias_mutate_rate        = 0.7
bias_replace_rate       = 0.1

# genome compatibility options
compatibility_disjoint_coefficient = 1.0
compatibility_weight_coefficient   = 0.5

# connection add/remove rates
conn_add_prob           = 0.5
conn_delete_prob        = 0.5

# connection enable options
enabled_default         = True
enabled_mutate_rate     = 0.01

feed_forward            = True
initial_connection      = full
```

Fig. 2. Arquivo de configuração NEAT.

```
initial_connection      = full

# node add/remove rates
node_add_prob           = 0.2
node_delete_prob        = 0.2

# network parameters
num_hidden              = 0
num_inputs              = 5
num_outputs             = 3

# node response options
response_init_mean      = 1.0
response_init_stdev     = 0.0
response_max_value      = 30.0
response_min_value      = -30.0
response_mutate_power   = 0.0
response_mutate_rate    = 0.0
response_replace_rate   = 0.0

# connection weight options
weight_init_mean        = 0.0
weight_init_stdev       = 1.0
weight_max_value        = 30
weight_min_value        = -30
weight_mutate_power     = 0.5
weight_mutate_rate      = 0.8
weight_replace_rate     = 0.1

[DefaultSpeciesSet]
compatibility_threshold = 3.0

[DefaultStagnation]
species_fitness_func = max
max_stagnation       = 20
species_elitism      = 2

[DefaultReproduction]
elitism            = 3
survival_threshold = 0.2
```

Fig. 3. Arquivo de configuração NEAT.

No proposto trabalho, será aplicado como função de ativação tangente, de modo que esta função de ativação permita-nos obter nosso valor para a camada de saída seja qual for o resultado e esmagar este valor em 2 valores definidos (-1;1). Desse modo, com o valor da saída será feita uma verificação de caso o valor seja menor de 0,5 neste caso o carro seguirá para o sentido esquerdo, senão para o sentido direito.

Na segunda fase é a implementação do código para a estruturação da IA juntamente com o jogo. Na função main, como mostrado na Fig 4, é possível analisar o caminho até o arquivo de configuração NEAT, a criação do core da evolução da classe do algoritimo e logo em seguida a parte da impressão no terminal dos status do melhor fitness, gerações. E logo em seguida é iniciado NEAT com o máximo de threshold.

```
__name__ == "__main__":

caminho = "./config-feedforward.txt"
config = neat.config.Config(neat.DefaultGenome,
neat.DefaultReproduction, neat.DefaultSpeciesSet,
neat.DefaultStagnation, caminho)

# Criando o core da evolução da classe do algoritmo
p = neat.Population(config)

p.add_reporter(neat.StdOutReporter(True))
stats = neat.StatisticsReporter()
p.add_reporter(stats)

# Rodando o NEAT
p.run(run_car, 1000)
```

Fig. 4. Função Main.

Na Fig 5 é feito os genomas com o tamanho da população que foi configurada no arquivo de texto. É preciso criar para cada rede o genomo e a fitness. E logo em seguida, é iniciado a classe do carro.

```
nets = []
##Lista dos carros
cars = []

## Irá fazer a inicialização dos genomas e a população
for id, g in genomes:
    net = neat.nn.FeedForwardNetwork.create(g, config)
    nets.append(net)
    g.fitness = 0
## Fim inicialização


    # Inicializando a classe dos carros
    cars.append(Car())
```

Fig. 5. Criação do Genomo e Fitness.

Na Fig 6 é onde cada carro irá passar pela rede, e com isso será pego a melhor ação atribuindo o angulo. Em seguida, é feita atualização do carro e do fitness onde o melhor carro, será recompensado com um valor definido, enquanto os outros não terão esta mesma sorte. Assim, podendo analisar a simplicidade na implementação de uma rede neural.

```
# ------------------ LOOP PRINCIPAL -----------------
global generation
generation += 1

while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit(0)

    # Inserindo os dados e pegando os dados da rede
    for index, car in enumerate(cars):## cada carro co

        ## mandar a posição dos radar (sensores) e det
        output = nets[index].activate(car.get_data())
        ##retorno da funcao de ativacao (tanh)
        i = output.index(max(output))

        ## O angulo que o carro irá realizar conforme
        if i <= 0.5:
            car.angle += 10
        else:
            car.angle -= 10

    # Atualizando o carro e o fitness
    carros_restant = 0
    for i, car in enumerate(cars):
        if car.vivo():
            carros_restant += 1
            car.atualizacao(map)
            genomes[i][1].fitness += car.recompensa()

# ------------------ FIM LOOP PRINCIPAL -------------
```

Fig. 6. Loop principal

Nesta terceira fase, será a implementação da renderização do jogo. É possível analisar na Fig 7 o mapa no qual o carro autônomo deverá andar sem que saia da rota/circuito. A inteligência artificial deverá conseguir pilotar em qualquer estrutura de circuito. Desse modo, foi feito 2 sensores na parte traseira do carro, 2 sensores na parte frontal do carro e um sensor na parte central totalizando 5 sensores. Com isso, no arquivo de configuração, num_imputs deverá ser 5, pois são os dados que irei colocar na rede.

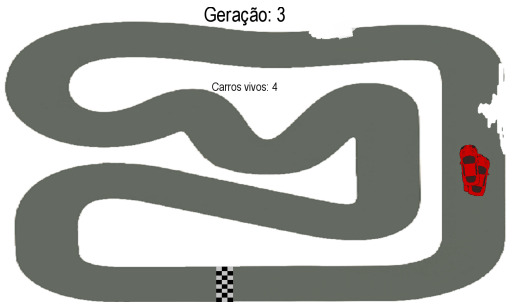

Fig. 7. Pista/Circuito

O carro deverá andar dentro da pista do circuito, desse modo foi feita a colisão como mostrado na Fig 8. A colisão, é ativada quando os sensores encostam na parede que foi configurada na cor branca.

```
def colisao_check(self, map):
    self.is_alive = True
    ##verificação da colisao dos 5 pontos direita esquerda top, direita
    for p in self.four_points:

        #verifcao para pegar o valor da cor de um unico pixel(x,y). Se
        #get_at (PYGAME)
        if map.get_at((int(p[0]), int(p[1]))) == (255, 255, 255, 255):
            self.is_alive = False
            break
```

Fig. 8. Colisão

A função destacada como atualizacao, encontra-se na próxima Figura. Nela é realizada a parte da locomoção do carro para que o mesmo consiga locomover-se de forma linear porém com os ângulos a serem proporcionados. Logo em seguida, é feita os cálculos das restrições da colisão, de forma que ao final atribua-se ao self.four_points para que o método colisao_check consiga chamar esta array e fazer a verificação da colisão.

```python
def atualizacao(self, map):
    #Velocidade
    self.speed = 15

    #Posição / andar
    self.rotate_surface = self.rot_center(self.surface, self.angle)
    self.pos[0] += math.cos(math.radians(360 - self.angle)) * self.speed
    if self.pos[0] < 20:
        self.pos[0] = 20
    elif self.pos[0] > screen_width - 120:
        self.pos[0] = screen_width - 120

    self.distance += self.speed
    self.time_spent += 1
    self.pos[1] += math.sin(math.radians(360 - self.angle)) * self.speed
    if self.pos[1] < 20:
        self.pos[1] = 20
    elif self.pos[1] > screen_height - 120:
        self.pos[1] = screen_height - 120

    # Calculo dos 4 pontos de colisao
    self.center = [int(self.pos[0]) + 50, int(self.pos[1]) + 50]
    ##tamanho do carro
    tamanho = 40
    esquerda_cima = [self.center[0] + math.cos(math.radians(360 - (self.angle + 30))) * tamanho, self.center[1]
    + math.sin(math.radians(360 - (self.angle + 30))) * tamanho]
    direita_cima = [self.center[0] + math.cos(math.radians(360 - (self.angle + 150))) * tamanho, self.center[1]
    + math.sin(math.radians(360 - (self.angle + 150))) * tamanho]
    esquerda_baixo = [self.center[0] + math.cos(math.radians(360 - (self.angle + 210))) * tamanho, self.center[1]
    + math.sin(math.radians(360 - (self.angle + 210))) * tamanho]
    direito_baixo = [self.center[0] + math.cos(math.radians(360 - (self.angle + 330))) * tamanho, self.center[1]
    + math.sin(math.radians(360 - (self.angle + 330))) * tamanho]
    self.four_points = [esquerda_cima, direita_cima, esquerda_baixo, direito_baixo]

    self.colisao_check(map)
    self.radars.clear()
    ##  faço for num range de -90 a 50 decremetando sempre 45. A distancia de quanto o carro vai conseguir ver pelos se
    for d in range(-90, 50, 45):
        self.check_radar(d, map)
```

Fig. 9. Atualização

IV. Resultados

No primeiro teste foi aderido uma população de 30 carros e um fitness de threshold 1000000, e com num_inputs de 5, pois representa os 5 sensores, juntamente com num_outputs de 2, pois são 2 saidas que podem ocorrer, como: virar para direita e para esquerda. Não foi aderido 3 saídas como a terceira sendo ”ir para frente”, de modo que o carro já está em constante andamento para frente. Com isso, os testes foram variados, foi possível em alguns casos o carro conseguir andar na pista em poucas gerações como mostrado na Fig 10 e na Fig 11 o resultado do fitness e o tempo.

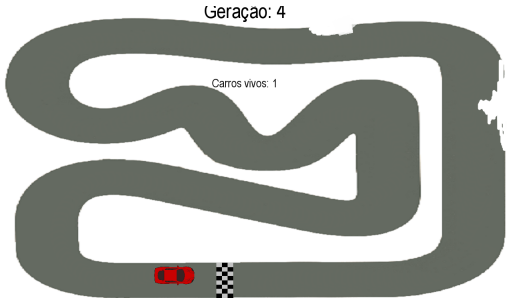

Fig. 10. Teste 1

```
****** Running generation 3 ******

Population's average fitness: 12409.00000 stdev: 31451.61330
Best fitness: 156600.00000 - size: (4, 12) - species 1 - id 106
Average adjusted fitness: 0.079
Mean genetic distance 1.467, standard deviation 0.255
Population of 30 members in 1 species:
   ID   age  size  fitness  adj fit  stag
  ====  ===  ====  =======  =======  ====
     1    3    30  156600.0    0.079     0
Total extinctions: 0
Generation time: 5.223 sec (3.577 average)
```

Fig. 11. Tempo e fitness 1

No segundo teste (Fig 12), foi aumentado a população para 50 carros, velocidaade em +5km/h e consequentemente o ângulo de virada também. Sem o aumento no ângulo, em algumas curvas seria impossível o carro conseguir virar a tempo de modo que não colidisse. A primeira volta foi completada na 8º geração, mas só apenas na 13º geração o carro autônomo iria completar o circuito de forma que não houvesse mais colisão. Na resolução da Fig 13, é possível observar o tempo e o fitness da geração 8.

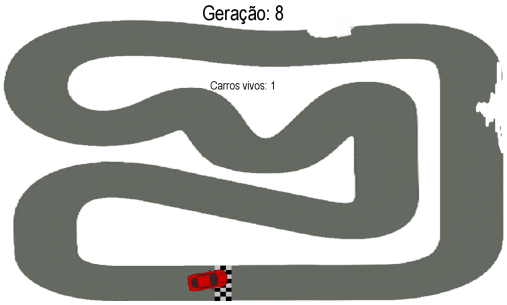

Fig. 12. Teste 2

```
****** Running generation 8 ******

Population's average fitness: 30472.00000 stdev: 84197.12052
Best fitness: 526700.00000 - size: (3, 13) - species 1 - id 316
Average adjusted fitness: 0.058
Mean genetic distance 1.086, standard deviation 0.344
Population of 50 members in 1 species:
   ID   age  size  fitness  adj fit  stag
  ====  ===  ====  =======  =======  ====
     1    8    50  526700.0    0.058     2
Total extinctions: 0
Generation time: 8.421 sec (6.894 average)
```

Fig. 13. Tempo e fitness Teste 2

V. Conclusão

Neste trabalho, o autor prôpos uma estratégia de treinamento utlizando NEAT para gerar carros autônomos capazes de dirigir sem haver algum tipo de colisão. O agente conseguiu dominar o circuito de modo que ao passar das gerações o algoritmo sempre pretende buscar a evolução mais implícita para um problema e que o fitness apresentado ajudou o NEAT a encontrar uma arquitetura de rede com a solução mais simples. As técnicas descutidas neste trabalho, auxiliaram o algoritmo a encontrar as soluções em um curto espaço de tempo, demonstrando sua eficácia.

Com o uso do tamanho da população de 30 carros e com velocidade não muito alta, foi possivel observar que o algoritmo evolucionário foi capaz de convergir para um estado ideal com poucas gerações. Com uma população de 50 e velocidade mais alta, foi observado que o algoritmo evolucionário demorou um pouco mais para chegar a um estado ideal. Neste ponto em ambos os testes, o agente poderá jogar o jogo infinitamente.

REFERÊNCIA

[1] POINTER, Ian. Conheça as 5 melhores linguagens de programação para Inteligência Artificial. 2018. Disponível em: https://computerworld.com.br/2018/07/04/conheca-5-melhores-linguagens-de-programacao-para-inteligencia-artificial/. Acesso em: 12 ago. 2020.

[2] T. Kristo and N. U. Maulidevi, "Deduction of fighting game countermeasures using Neuroevolution of Augmenting Topologies," 2016 International Conference on Data and Software Engineering (ICoDSE), Denpasar, 2016, pp. 1-6, doi: 10.1109/ICODSE.2016.7936127.

[3] THIHA, Phyo. Extending Robot Soccer Using NEAT. 2009. Disponível em: https://www.cs.swarthmore.edu/ meeden/cs81/s09/finals/Phyo.pdf. Acesso em: 13 ago. 2020.

[4] S. Samothrakis, D. Perez-Liebana, S. M. Lucas, and M. Fasli, "Neuroevolution for General Video Game Playing," 2015 IEEE Conference on Computational Intelligence and Games, CIG 2015 - Proceedings, pp. 200–207, 2015.

[5] NEAT-PYTHON. Configuration file description. Disponível em: https://neat-python.readthedocs.io/en/latest/config_file.html. Acesso em: 12 ago. 2020.

[6] CORDEIRO, Matheus G.; SERAFIM, Paulo Bruno S.; NOGUEIRA, Yuri Lenon B.; VIDAL, Creto A.; CAVALCANTE NETO, Joaquim B.. A Minimal Training Strategy to Play Flappy Bird Indefinitely with NEAT. 2019. Disponível em: https://ieeexplore.ieee.org/abstract/document/8924807. Acesso em: 13 ago. 2020.

[7] J. Schrum and R. Miikkulainen, "Discovering multimodal behavior in ms. pac-man through evolution of modular neural networks," IEEE Transactions on Computational Intelligence and AI in Games, vol. 8, no. 1, pp. 67–81, March 2016.

[8] J. Schrum and R. Miikkulainen, "Constructing complex npc behavior via multi-objective neuroevolution," in Proceedings of the Fourth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, ser. AIIDE'08. AAAI Press, 2008, pp. 108–113. [Online]. Available: http://dl.acm.org/citation.cfm?id=3022539.3022558

[9] M. Wittkamp, L. Barone and P. Hingston, "Using NEAT for continuous adaptation and teamwork formation in Pacman," 2008 IEEE Symposium On Computational Intelligence and Games, Perth, WA, 2008, pp. 234-242, doi: 10.1109/CIG.2008.5035645.