AI joga Flappy Birds

José Montes, Leonardo Thurler, Rodrigo Veloso
Universidade Federal Fluminense

Resumo

Uma formulação para o jogo Flappy Bird foi desenvolvida com o objetivo de criar um agente capaz de obter uma alta pontuação de forma consistente ou nunca perder no jogo. Foram considerados dois tipos principais de metodologias de inteligência artificial para resolver o problema: busca e aprendizado por reforço.

Considerando a abordagem de busca, foi aplicado o algoritmo NEAT, um algoritmo genético que considera que indivíduos de uma população são redes neurais artificiais que através de uma função de fitness busca-se pela melhor combinação de pesos e arquitetura para a solucionar o problema. No contexto de aprendizado por reforço dois algoritmos foram considerados: Q Learning, um algoritmo clássico baseado na função valor, e PPO, um algoritmo moderno de aprendizado por reforço profundo.

Diferentes representações de estado e funções de recompensa e fitness foram desenvolvidas e testadas empiricamente para cada uma das abordagens. Não foi possível determinar uma melhor função ou representação para o problema, cada algoritmo se saiu melhor em uma configuração de estado e função diferente. Foram também realizadas buscas por parâmetro ótimos de cada problema. Após a busca um teste final foi realizado, onde foi identificado que o agente com melhor resultado foi o treinado utilizando Q Learning.

Todos os algoritmos conseguiram realizar a tarefa de jogar o jogo e obter uma alta pontuação, superando a performance humana para um mesmo tempo de treinamento.

Introdução

A inteligência artificial(I.A.) é considerada um dos campos mais novos da ciência, tendo seus primeiros trabalhos realizados na década de 50, cujo principal foco é a reprodução do pensamento humano em máquinas. (Soder and Malheiros, 2019).

Apesar de soluções de Inteligência Artificial existirem desde a década de 50, apenas nos últimos anos diversas técnicas que podiam ser realizadas apenas em laboratórios especializados agora podem ser realizados em computadores pessoais graças ao avanço do poder computacional de dispositivos pessoais, democratizando o acesso à tais tecnologias a um mundo antes restrito a poucos (Soder and Malheiros, 2019). Além do aumento do poder computacional, outro fator está ligado à democratização de técnicas de inteligência artificial: a disponibilidade de dados existentes, uma vez que soluções de IA geralmente precisam de muitos dados para treinamento.

Com o avanço tecnológico ao longo dos anos, a complexidade dos problemas computacionais e o volume de dados gerados vem crescendo cada vez mais. Devido a isso, vem se tornando cada vez mais claro a necessidade de termos ferramentas computacionais que atuem de forma mais autônomas reduzindo a necessidade de intervenção humana e de especialistas. Para atingir esse objetivos, é desejado que essas ferramentas sejam capazes de, através de experiências passadas, criarem uma hipótese capaz de resolver o problema que se deseja tratar. (Faceli et al., 2011)

Nesse cenário, a sub-área de Aprendizado de Máquina tem se mostrado bastante relevante justamente por nos fornecer os recursos desejados. Ao longo dos últimos anos passou a possuir aplicações práticas bem-sucedidas em diversas áreas, como Reconhecimento de palavras faladas, predição de taxas de cura de pacientes com diferentes doenças, condução de automóveis de forma autônoma em rodovias, ferramentas que jogam jogos de tabuleiro de forma semelhante a campeões.

O objetivo final do trabalho consiste em realizar uma analise e comparação de 3 diferentes algoritmos, sendo 1 de busca e os outros 2 de aprendizado de máquina. Para isso foi desenvolvido um ambiente de aprendizado aplicado ao jogo Flappy Birds, este ambiente utiliza o NEAT como algoritmo de busca que se baseia na utilização de algoritmo genético com redes neurais, também foi utilizado o Q-Learning como algoritmo de aprendizado por reforço, e por fim, o PPO como algoritmo de aprendizado por reforço profundo. Para a construção do ambiente de aprendizado, foram utilizadas 2 engines de criação de jogos digitais a Unity e a Pygame.

Fundamentação Teórica

Redes Neurais Artificiais

A utilização de Redes Neurais Artificiais têm se destacado como solução de I.A. nos últimos anos devido a

sua aplicação a problemas cada vez mais complexos, uma vez que o seu processo de treinamento é capaz de extrair informações relevantes de padrões de entrada através de simulações cognitivas, o que permite realizar predições em quase todos os problemas práticos do cotidiano (Furtado and Furtado, 2020).

Uma Rede Neural Artificial (RNA) é composta de uma unidade elementar chamada de neurônio artificial, alocados em camadas interconectadas através de um grande número de conexões com o objetivo de calcular funções matemáticas (Funções de Ativação). Cada uma dessas conexões está ligando uma camada à outra camada adjacente, as conexões possui um peso, o que impacta na saída do neurônio no resultado final da RNA, onde quanto maior o peso, maior o impacto daquela ligação na saída resultante daquela rede. O processo de treinamento se dá através do ajuste desses pesos.

Neurônio Artificial Dentro de uma Rede Neural Artificial, o neurônio é a unidade elementar de processamento, recebendo sinais de entrada através de uma camada de entrada ou da saída de outros neurônios anteriores, que são multiplicados pelos pesos de cada ligação e somados através de uma função somatória como é possível observar na Figura 1 (Alves, de Carvalho, and de A Sabino, 2020), em que está demonstrado o esquema de processamento do Perceptron, primeiro tipo de neurônio artificial (Haykin, 2007).

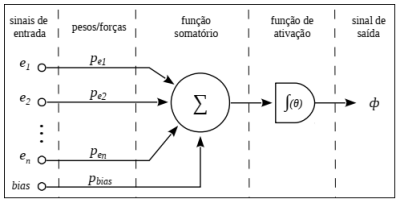

Figura 1: Unidade básica de processamento.
(Alves, de Carvalho, and de A Sabino, 2020)

Após a multiplicação das entradas pelos pesos e a sua somatória, o resultado é utilizado em uma função de ativação, que será utilizado para normalizar as saídas de cada neurônio.Diversas funções de ativação podem ser utilizadas, sendo a sigmóide amplamente utilizada na solução de problemas diversos em RNAs, porém redes neurais contendo apenas um único Perceptron (Neurônio Artificial) pode resolver apenas problemas muito simples, não tendo muitas aplicações de ordem prática, sendo necessário a utilização de diversas camadas de neurônios para a solução de problemas difíceis, surgindo assim a Rede Neural Artificial Multicamada, também chamada de MultiLayer Perceptron (MLP) (Haykin, 2007).

MultiLayer Perceptron Múltiplas camadas intermediárias podem ser utilizadas para dar maior flexibilidade à rede na adequação da sua estrutura na treinamento de cenários para problemas mais complexos e que exigem maior processamento de dados, uma vez que RNAs de uma única camada não conseguirão bons resultados para a maioria desses problemas (Alves, de Carvalho, and de A Sabino, 2020).

A estrutura presente no Perceptron de única camada ainda se mantém a mesma (Entrada, Pesos, Somatória, Função de Ativação e Saída), porém são utilizados várias camadas contendo vários neurônios, como pode ser demonstrado na Figura 2

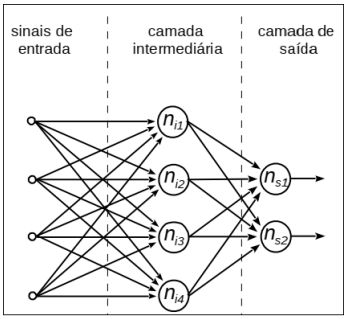

Figura 2: Exemplo de Rede Neural Artificial Multicamada.
(Alves, de Carvalho, and de A Sabino, 2020)

Após a definição da sua arquitetura e topologia (Número de Camadas e Número de Neurônios em cada Camada, também chamados de Hiper parâmetros da Rede Neural), o seu treinamento se dá de forma supervisionada, onde para cada entrada da rede é determinado a sua saída desejada, sendo esse tipo de rede neural aplicado na solução de diversos problemas difíceis (Haykin, 2007).

O treinamento de uma rede neural geralmente é formulado como uma minimização de uma função de erro entre as saídas reais da rede e as saídas desejadas de todos os padrões de treinamento, através de um ajuste iterativo dos pesos, sendo o BackPropagation o algoritmo de aprendizado mais utilizado (YAMAZAKI, 2004). Considerando a natureza da otimização proporcionado pelo treinamento de uma rede neural, é possível a aplicação de técnicas de busca para encontrar uma solução de pesos a fim de minimizar a sua função de erro.

Considerando essa possibilidade o presente trabalho irá apresentar também a utilização de um método de busca para otimização dos pesos de uma rede neural. O método de busca utilizado foi o algoritmo genético, uma vez que eles foram concebidos inicialmente para realização de busca mais geral no espaço (YAMAZAKI, 2004), procurando um ponto mínimo de erro para a rede neural artificial levando conta aspectos globais da superfície de erro.

Algoritmo Genético

O algoritmo genético (AG) constitui um modelo matemático simulando a teoria da evolução Darwiniana, no qual consiste inicialmente em um conjunto de possíveis soluções para um determinado problema, chamada de população inicial, sendo esta criada de forma aleatória na maioria das aplicações. (LACERDA et al., 1999) Gonçalves (2019) (Marsland, 2009)

A implementação de Algoritmos Genéticos também são inspirados em conceitos da genética, utilizando alguns recursos como codificação de cromossomos, função de aptidão ou avaliação (fitness), processo de seleção, além das

etapas de cruzamento e mutação, e da própria evolução das populações (Soder and Malheiros, 2019)

Uma nova população de soluções (indivíduos) é gerada através da aplicação de operadores genéticos como seleção, cruzamento e mutação com o intuito de explorar regiões desconhecidas do espaço de busca de forma global, sendo esse processo repetido até que o indivíduo mais apto seja encontrado ou uma condição de parada seja satisfeita, sendo esta geralmente o número de gerações. Após a finalização, o indivíduo mais apto, ou seja, com a maior avaliação dentre os existentes é dito ser a solução do problema, tendo assim um AG genético convergido, como pode ser observado na Figura 3 (LACERDA et al., 1999) (Lopes, 2006).

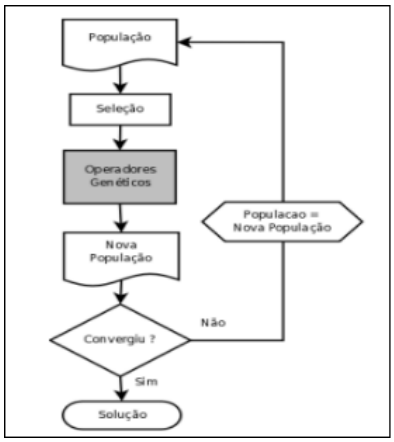

Figura 3: Ciclo de vida de um Algoritmo Genético. (Gonçalves, 2019)

Segundo (Soder and Malheiros, 2019) cada solução (indivíduo) de um AG é composta de um cromossomo, abstração de um DNA biológico contendo valores numéricos com a solução do problema procurado e o seu nível de aptidão como solução do problema (fitness). No processo evolutivo, o nível de aptidão de um cromossomo é utilizado como ferramenta de seleção de uma geração para outra, fazendo com que os indivíduos mais aptos (Melhores Soluções) tenham maiores chances de serem escolhidos para recombinação e estarem nas próximas gerações (Gonçalves, 2019)

Considerando a natureza do presente trabalho, o processo de treinamento de uma Rede Neural Artificial (RNA) utilizando um Algoritmo Genético (AG)

> ”pode ser visto como um problema de otimização onde o espaço de busca é o espaço de pesos da rede e a função objetivo é uma função do desempenho obtido pela rede para um determinado conjunto de padrões” (Prudêncio, 2001)

Onde cada indivíduo do AG foi composto de uma RNA diferente como forma de solução para o problema proposto pelo artigo: Implementar uma solução de Inteligência Artificial que seja capaz de jogar o jogo Flappy Bird. Foi utilizado o algoritmo de NeuroEvolução de Topologias Aumentantes NEAT para implementação da busca usando Algoritmo Genético com a finalidade de encontrar a melhor combinação de pesos entre as ligações dos neurônios e ajustar também as próprias ligações entre os neurônios.

NEAT Uma característica bastante impactante no desempenho de uma Rede Neural é a sua Topologia ou definição de Hiperparâmetros, como quantidade de camadas, quantidade de neurônios em cada camada, funções de ativação, épocas de treinamento entre outras características que, geralmente, são definidas de forma empírica e demandam muito tempo de testes e ajustes manuais, muitas vezes ainda sem conseguir atingir, de fato, uma configuração ótima para problemas envolvendo redes neurais. Considerando isso, métodos de busca podem ser utilizados na otimização dos parâmetros de uma rede neural, sendo essa uma das premissas de funcionamento do algoritmo NEAT.

O algoritmo NEAT utiliza algoritmo genético para otimização de redes neurais, tanto dos seus pesos, realizando o processo de treinamento, quanto da sua arquitetura, com a finalidade de atingir um bom resultado com a menor arquitetura de rede neural possível (Stanley and Miikkulainen, 2002)

O algoritmo NEAT codifica uma rede neural na forma de algoritmo genético, onde cada indivíduo representa uma rede neural. O cromossomo de cada indivíduo possui dois tipos de gene: Neurônio e Ligação. Cada neurônio codificado possui apenas o seu código e o tipo de neurônio (Entrada, Oculto ou Saída) e cada ligação possui os neurônios que estão conectados, o seu peso e se essa ligação está ativa ou não, como representado na Figura 4

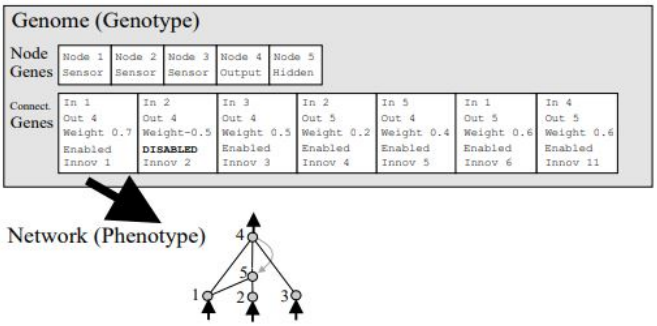

Figura 4: Codificação RNA - Cromossomo. (Stanley and Miikkulainen, 2002)

O processo de busca implementado pelo algoritmo NEAT envolve a utilização dos operadores genéticos de Seleção, Cruzamento e Cruzamento, implementados da seguinte forma:

- Seleção: Sempre que uma nova população é gerada, os indivíduos da geração anterior são selecionados para aplicação dos operadores de Cruzamento e Mutação para criação da próxima geração (Com exceção da primeira geração que é gerada de forma aleatória). A seleção dos indivíduos é feita utilizando uma função Fitness para cada indivíduo que avalia a qualidade de uma solução para um determinado problema através de um indicador numérico.
- Cruzamento: Dois indivíduos são selecionados e combi-

nados entre si a fim de se criar um novo indivíduo com uma melhor função de avaliação

- Mutação: Processo no qual alguma informação existente no Gene é alterada de forma aleatória, geralmente com uma chance muito baixa, com a finalidade de aumentar o espaço de busca, que podem ser aplicado nos pesos das ligações entre os neurônios e nas próprias ligações além da possibilidade de adicionar novos neurônios no indivíduo.

O processo de busca implementado pelo NEAT possui dois objetivos: diminuir a função de erro da rede neural e encontrar a solução mais simples, dessa forma, é definida a topologia inicial da rede que vai ser utilizada como base para a criação de todos os indivíduos da geração inicial,

Aprendizado por Reforço

Podemos entender aprendizado por reforço (RL) como a interação entre um agente e um ambiente. O agente é quem aprende, e o ambiente é onde o aprendizado ocorre. O ambiente é responsável de fornecer ao agente a informação sobre a qualidade da estratégia do agente através de um função de recompensa (Marsland, 2009).

O agente sob este tipo de aprendizado deve experimentar diferentes estratégias para descobrir qual funciona melhor. O processo de experimentar diferentes estratégias é uma outra forma de descrever o processo de busca, como visto na seção anterior. O agente busca em todo espaço de possíveis entradas e saídas (estado/ação) na tentativa de adquirir uma recompensa.

O aprendizado por reforço mapeia um estado à uma ação, para tentar maximizar uma recompensa, ou seja, o agente conhece o estado atual em que se encontra e as possíveis ações que pode tomar e com essas informações tenta conseguir o máximo de recompensa possível.

Um processo genérico de RL segue os seguintes passos: primeiramente o agente se encontra em um estado s_t em um instante de tempo t, o a gente então realiza uma ação a_t, esta ação causa uma transição do estado s_t para o estado s_{t+1} e então o agente recebe uma recompensa r_t. O processo descrito define uma experiência $< s_t, a_t, s_{t+1}, r_t >_t$, o agente então repete esse processo um número determinado de vezes, observando um número determinado de experiências e conforme o número de experiências aumentam é esperado um aumento na recompensa obtida pelo agente, portanto, um aumento de desempenho. Essa metodologia considera o tempo como uma variável discreta (Marsland, 2009).

Diferentemente do aprendizado supervisionado, onde se é ensinada a resposta certa ao algoritmo, a função de recompensa avalia a resposta mas não ensina como melhorar, o que naturalmente torna o processo de aprendizado mais difícil. Outra dificuldade é a possibilidade da recompensa ser adiada, ou seja, só há informação sobre a qualidade da solução encontrada após um intervalo de tempo, que pode ser longo. É então importante pensar em recompensas imediatas mas também na recompensa esperada no futuro.

Uma vez definidas as recompensas, é preciso escolher a ação que será executada no estado atual. A função que mapeia um estado em uma ação, e portanto responsável pela escolha da ação, é conhecida como política (π). A política é em geral baseada em uma combinação entre *exploitation* e *exploration*, ou seja, escolher a ação que resultou na maior recompensa no passado ou uma ação diferente na esperança de obter uma recompensa ainda melhor.

O objetivo de um agente é então aprender a ação que maximiza a recompensa esperada para todo possível estado, aprender a política ótima. A política ótima π^* é aquela que maximiza a esperança da soma descontada das recompensas, ou seja, π^* é tal que:

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\, Q^*(s,a) \tag{1}$$

onde $Q^*(s,a)$ é a função valor estado ação ótima, que pode ser definida por

$$Q^*(s,a) = E(\sum_{t=0}^{\infty} \gamma_t r(s,a)), \tag{2}$$

onde γ é o fator de desconto, parâmetro que define o quanto o agente valoriza as recompensas imediatas em relação às recompensas futuras e $E(.)$ é a esperança estatística.

Q Learning A função valor estado ação (Q) mapeia um conjunto (s,a) ao valor de uma recompensa esperada

$$Q(s,a) = E(r_t | s_t = s, a_t = a). \tag{3}$$

Se a função Q é conhecida é fácil determinar a política ótima, basta considerar uma política gulosa, onde basta sempre tomar a ação que maximiza o valor de Q em um determinado estado. O problema é que a função Q não conhecida apriori (Marsland, 2009).

O algoritmo Q Learning busca aprender a função valor estado ação (Q), através de um processo iterativo. A ideia é criar uma tabela com valores de Q para todos os possíveis pares estado ação e ir atualizando a tabela a partir das experiências. A tabela pode ser iniciada de forma aleatória ou com todos os valores iguais à zero. A atualização da tabela é feita utilizando a equação

$$Q(s_t, a_t)_{i+1} = Q(s_t, a_t)_i + \mu_i(r_t + TD_i) \tag{4}$$

onde μ_t é a taxa de aprendizado e TD_i é a diferença temporal dada por

$$TD_i = \gamma_t max_{a_{t+1}} Q_i(s_{t+1}, a_{t+1}) - Q_i(s_t, a_t). \tag{5}$$

Como a tabela é inicializada de forma aleatória, a função Q_i se aproxima de Q de forma iterativa e não há garantia de convergência para a Q real do problema, sendo assim, escolher sempre a ação que maximiza Q_i não resultará na política ótima, principalmente durante o início do treinamento, onde Q_i está longe de convergir. Durante o treinamento uma política gulosa irá beneficiar a *exploitation* mas não irá realizar nenhum tipo de *exploration*, o que é indesejável.

Existem algumas formulações de políticas para introduzir um maior exploração do espaço e consequentemente identificar estratégias mais inteligentes. Uma política bastante

conhecida e utilizada é a ϵ-greedy (Marsland, 2009). A ϵ-greedy é similar a estratégia gulosa, onde escolhe a tal que $Q(s,a)$ é máxima, mas com uma pequena probabilidade ϵ é escolhida uma ação diferente aleatoriamente. A política ϵ-greedy encontrará soluções melhores que a política puramente gulosa pois é possível fazer *exploration*. O parâmetro ϵ pode variar com o tempo, no ínicio do processo de aprendizado há pouco ou nenhum conhecimento sobre o problema, sendo então mais interessante realizar mais *exploration* utilizando maiores valores de ϵ, com o aumento do número de experiências observadas pelo agente há uma maior confiança na Q aprendida, sendo então mais interessante realizar mais *exploitation* utilizando ϵ menores. Considerar ϵ como uma função que decresce com o tempo é altamente recomendável.

Tsitsiklis (1994) demonstrou a convergência do algoritmo Q learning. Para garantir a convergência desse algoritmo são necessárias algumas condições, uma delas diz respeito ao parâmetro μ. É preciso que μ respeite as condições

$$\sum_{i=0}^{\infty} \mu_i = \infty \; e \; \sum_{i=0}^{\infty} \mu_i^2 \leq C, \quad C < \infty. \tag{6}$$

As condições anteriores indicam que μ deve ser inversamente proporcional ao número de iterações. É natural pensar que a taxa de aprendizado deve diminuir com o tempo, no início do aprendizado as mudanças podem ser feitas de forma mais rápida mas com o passar do tempo um alta taxa de aprendizado pode atrapalhar a convergência.

O algoritmo 1 é o algoritmo do Q Learning.

inicializa $Q(s,a)$ com números aleatórios pequenos ou zero para todo s e a;
while $i < max\_i$ **do**
  inicia s;
  while *houverem episódios* **do**
    escolha a usando alguma política π;
    tome a e receba r;
    obtem o novo estado s';
    $Q(s,a) \leftarrow Q(s,a) + \mu(r + \gamma max_{a'} Q(s',a') - Q(s,a))$;
    $s \leftarrow s'$;
  end
  $i \leftarrow i + 1$;
end

Algorithm 1: Q Learning

A maior dificuldade do algoritmo Q Learning está no fato da necessidade de armazenar todos os valores de Q para todo par (s,a) em uma tabela. Quanto maior for o espaço de estados ou ações, maior a memória necessária para armazenar a tabela, limitando assim as possíveis representações de estado à quantidade de memória disponível. Em casos onde o espaço de estados e ações é muito extenso ou contínuo é impossível a utilização do Q Learning. Uma possível solução é a discretização do espaço de estados ou a utilização de representações mais inteligentes, que ocupam menos memória.

Aprendizado por Reforço Profundo

Aprendizagem profunda, do inglês Deep Learning (DL), é uma estratégia de aprendizagem que busca permitir que computadores aprendam através de uma hierarquia de conceitos, onde cada conceito pode ser definido através de uma relação com outro conceito mais simples. (Goodfellow, Bengio, and Courville, 2016)

No aprendizado profundo, temos as camadas de entrada, saída e uma ou mais camadas ocultas. Com exceção da camada entrada, os dados de entrada de cada camada é gerado através da soma dos pesos das camadas anteriores. Deste modo, a DL consegue aprender representações de forma automática a partir de entradas brutas para recuperar as hierarquias de composição em muitos sinais naturais. (Li, 2018)

Diversos algoritmos que fazem da estratégia de DL, utilizam redes neurais artificiais (NN) para a implementação de DL. A Figura 5 apresenta um exemplo de rede neural com 2 camadas escondidas contendo 3 neurônios em cada.

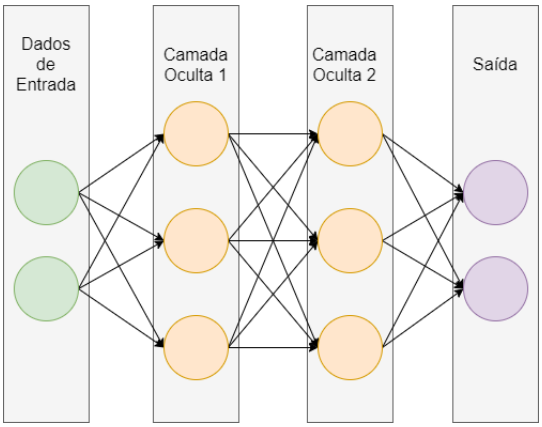

Figura 5: Exemplo de Rede Neural Artificial com 2 camadas escondidas.

O aprendizado por reforço profundo (DRL) busca juntar as estratégias de RL com DL. Algoritmos com essas características tem apresentados resultados bastante impressionante. Um exemplo é o AlphaGO (Silver et al., 2016), que foi desenvolvido por pesquisadores da empresa Google DeepMind, este algoritmo foi capaz de vencer Lee Sedol, jogador profissional de 9º Dan que foi 18 vezes campeão mundial, no jogo de tabuleiro Go.(AlphaGo versus Lee Sedol)

PPO Otimização de Política Proximal, em inglês Proximal Policy Optimization (PPO), é um algoritmo proposto por Schulman et al. (2017) que busca otimizar o desempenho de estratégias de RL em ambientes que tenham ações e observações contínuas ou com uma grandeões. O PPO se baseia em uma arquitetura chamada Actor-Critic (Atuação-Avaliação). Essa arquitetura utiliza 2 redes neurais, a primeira é chamada de critic e é utilizada para avaliar o estado atual do ambiente gerando uma estimativa de valor Q, a segunda serve para definir a melhor ação para o estado atual, esta é chamada de actor. A Figura 6 apresenta um diagrama exemplificando esta arquitetura. (Pecenin, 2019)

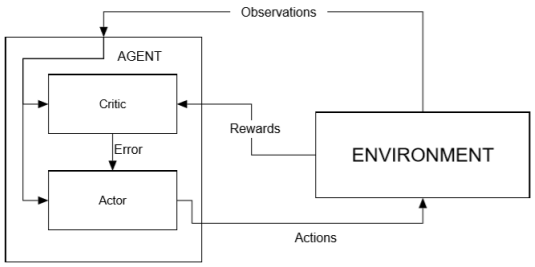

Figura 6: Exemplo de uma arquitetura actor-critic (Fig. do Lanham (2018))

Realizando uma comparação direta com o Q Learning, este algoritmo utiliza a função de Vantagem (Ad) ao invés da função valor estado ação $Q(s, a)$. É na verdade calculada uma estimativa da recompensa que depende somente de s, essa estimativa é mais genérica por não depender de uma ação específica como a função $Q(s, a)$. Utilizando a estimativa de recompensa menos a média da função de valor no estado $s(V(s))$, é possível representar a função de Vantagem através da equação 7. Uma outra vantagem do PPO usar a arquitetura actor-critic, está no fato desta arquitetura ter sido desenvolvida para trabalhar com vários agentes assíncronos, onde cada agente tem seu próprio ambiente, o que resulta em um processo de treinamento mais ágil. (França, 2019; Pecenin, 2019)

$$Ad = r - V(s) \tag{7}$$

A rede de atuação actor representa a política do agente para escolha de ações. De acordo com Pecenin (2019), isso significa que na prática, esta rede representa o estado s mapeado para uma função gaussiana ou categórica de distribuição de probabilidade, de onde será retirado o valor efetivo da ação através do uso do valor médio e desvio padrão da distribuição. A figura 7 exemplifica esse processo.

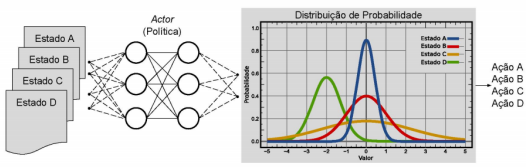

Figura 7: Exemplo da política de escolha de ações do PPO, (Fig. do Pecenin (2019))

A figura 7 apresentou uma comunicação de informações de erro entre as redes critic e actor, esse erro pode ser representado pelas equações 8 e 9.

$$ValueLoss = \sum (R - V(s))^2 \tag{8}$$

$$PolicyLoss = -log(\pi(a|s)) * Ad(s) \tag{9}$$

O algoritmo busca minimizar esses erros através do cálculo de entropia. De acordo com Lanham (2018), a entropia mede a propagação de probabilidade, enquanto uma entropia alta representa um agente com várias ações similares, o que dificulta as decisões do agente, uma entropia baixa nos diz que um agente pode tomar decisões mais bem informadas. Uma explicação mais detalhada pode ser encontrada no artigo original do PPO. Schulman et al. (2017)

Descrição e Modelagem do Problema

Flappy Bird

Flappy Bird é um jogo eletrônico para um jogador, onde o jogador controla um único pássaro com objetivo de obter a maior pontuação possível, o jogo dá um ponto toda vez que o pássaro passa entre dois canos, um cano superior e um cano inferior.

O jogador tem apenas duas ações possíveis: pular, onde o pássaro terá sua posição vertical (y_p) deslocada para cima, e fazer nada, onde o pássaro irá se deslocar para baixo devido a ação da gravidade implementada no jogo. A posição horizontal do pássaro (x_p) é sempre a mesma, a posição horizontal dos canos (x_c) é modificada considerando um movimento com velocidade constante. A posição vertical da extremidade do cano inferior (x_{ci}) e do cano superior (x_{cs}) são definidas de forma aleatória mantendo sempre um espaçamento constante entre os dois. Quando o pássaro passa entre os dois canos, dois novos canos são gerados.

Se o jogador permitir que o pássaro colida com o chão ou com algum dos canos o jogo é encerrado e é então atribuída a pontuação total obtida até o momento da colisão. Um novo jogo pode então ser iniciado onde a pontuação inicial é igual à zero. Para se obter uma alta pontuação é necessário então que o jogador mantenha o pássaro sem colidir pelo maior tempo possível. Uma versão do jogo está disponível em `http://flappybird.io/` e a figura 8 ilustra os elementos principais do jogo.

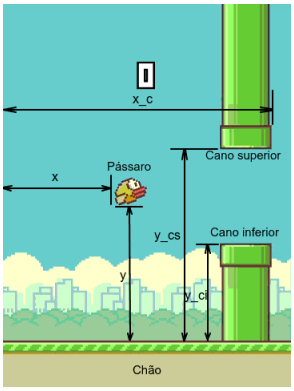

Figura 8: Ilustração do jogo Flappy Bird.

Tomando como base a versão citada anteriormente, foram implementadas duas versões do jogo: uma desenvolvida utilizando a Unity e a outra utilizando Pygame. Ambas as versões foram implementadas buscando uma reprodução fiel da versão base sempre quando possível. Foram utilizadas duas implementações devido a maior facilidade de

implementação do algoritmo PPO na Unity e maior facilidade do algoritmo Q Learning em Pygame.

Um modelo para o Flappy Bird

Criar um agente que consiga jogar o Flappy Bird e nunca perder ou que consiga obter um bom desempenho de forma consistente pode ser formulado como um problema que pode ser resolvido aplicando metodologias de inteligência artificial.

É um problema de agente único, pois o desempenho do agente não depende e nem é afetado por nenhum outro, o jogo é de jogador único e só há um pássaro por vez.

O ambiente é completamente observável, é possível coletar todas as informações relevantes para o agente conseguir decidir qual ação tomar. O ambiente não é determinístico pois não é possível saber a prior a posição dos canos que o pássaro deverá passar. É um ambiente episódico, onde episódios futuros não dependem dos episódios passados. Apesar de ser um jogo dinâmico o ambiente é estático pois o ambiente não muda entre o tempo de percepção e ação, as ações e percepções são tomadas no mesmo frame. A implementação do jogo na Unity considera um ambiente contínuo enquanto a do Pygame considera um ambiente discreto.

Modelagem do Ambiente O ambiente pode ser modelado de diversas formas, é possível considerar o ambiente como o conjunto de pixeis que aparecem na tela, por exemplo. No entanto, o jogo é bem simples e possui baixa dimensionalidade, um estado pode ser modelado considerando apenas a posição do pássaro e dos canos sendo possível definir a posição do pássaro utilizando uma componente de velocidade vertical v e sua posição y e a posição dos canos utilizando uma componente de velocidade horizontal v_c constante e posições y_{cs}, y_{ci} e x_c. É considerado então o modelo mais simples, utilizando um vetor de 4 posições para definir o estado $s = (y, y_{cs}, y_{ci}, x_c)$.

Modelo de Transição e Meta O agente pode executar duas possíveis ações, a ação de pular e a ação de fazer nada (não pular). A partir dessas ações é possível definir um modelo de transição para o problema:

$$\text{result}((y, y_{cs}, y_{ci}, x_c), \text{pular}) = (y + p, y_{cs}, y_{ci}, x_c + v_c) \tag{10}$$

$$\text{result}((y, y_{cs}, y_{ci}, x_c), \text{não pular}) = (y - g_t, y_{cs}, y_{ci}, x_c + v_c) \tag{11}$$

onde p é o deslocamento causado pelo pulo e g_t é o deslocamento causado pela gravidade, que varia com o tempo. Existem duas outras possíveis transições que só ocorrem quando $x_c = x$, $y < y_{cs}$ e $y > y_{ci}$, ou seja, no momento em que o pássaro atravessa os canos:

$$\text{result}((y, y_{cs}, y_{ci}, x_c), \text{pular}) = (y + p, y'_{cs}, y'_{ci}, x_{max}) \tag{12}$$

$$\text{result}((y, y_{cs}, y_{ci}, x_c), \text{não pular}) = (y - g_t, y'_{cs}, y'_{ci}, x_{max}) \tag{13}$$

onde y'_{cs} e y'_{ci} são as novas posições dos canos, definidas de forma aleatória e x_{max} é posição horizontal máxima.

O jogo é encerrado no momento de uma colisão com o chão ($y \leq y_{min}$) ou com algum dos canos.

Método

Métricas de desempenho

Todos os métodos que serão avaliados e comparados neste trabalho são métodos iterativos. Métodos iterativos buscam obter soluções através de melhorias sucessivas de uma estimativa inicial até que um número máximo de iterações seja atingido ou até que o desempenho da solução seja superior à um limite inferior pré estabelecido.

É importante então obter formas objetivas de avaliar as soluções encontradas ao longo do processo iterativo, para isso é preciso definir métricas de desempenho. As métricas de desempenho são também necessárias pois possibilitam a comparação entre diferentes agentes e metodologias de forma objetiva.

Em aprendizado por reforço uma métrica interessante e que não depende do problema é a recompensa acumulada do agente, se um agente obtém maiores recompensas a medida em que observa mais experiências e função de recompensa foi bem definida, é garantido que seu desempenho está aumentando, independente do problema que se deseja resolver. Para comparar dois agentes utilizando a recompensa acumulada é preciso garantir que os dois possuam a mesma função de recompensa ou que haja uma conversão entre as duas.

Pode-se também definir métricas específicas para um determinado problema, dessa forma é possível comparar qualquer metodologia. No contexto do Flappy Bird, o objetivo do agente é passar pelo maior número de canos possíveis para obter a maior pontuação possível, é natural então medir o desempenho de um agente através da pontuação que ele obtém. Como o Flappy Bird não é determinístico e o aprendizado por reforço é um processo que depende de tentativa e erro, é natural que haja uma oscilação na performance do agente, mesmo que este já esteja treinado, por isso é importante considerar a pontuação média obtida.

O desempenho de um agente pode ser avaliado através de sua consistência em realizar uma tarefa pre determinada. Se a tarefa do agente é passar por dois canos, por exemplo, é possível então contar em quantos jogos essa tarefa foi realizada e calcular a probabilidade do agente realizar a tarefa. Quanto maior a probabilidade do agente realizar a tarefa, mais consistente é sua estratégia e portanto maior será seu desempenho.

Neste trabalho as principais métricas utilizadas para comparar agentes e avaliar seus desempenhos serão a recompensa acumulada, a pontuação média, máxima e a probabilidade da travessia de dois canos.

Observação dos estados

Apesar de os estados serem definidos usando o vetor (y, y_{cs}, y_{ci}, x_c), a observação do agente pode ser feita considerando menos variáveis ou variáveis diferentes. Utilizar o vetor de estado como o estado observado pelo agente nem sempre é a melhor estratégia, no algoritmo Q Learning, por

exemplo, devem ser utilizadas apenas variáveis discretas e com um limite de possibilidades baixo, o que não é o caso do vetor de estado original.

Uma solução para reduzir o espaço de estados é considerar uma discretização das variáveis, sem uma discretização é impossível aplicar o Q Learning. A forma de discretização escolhida é dada pela equação

$$var_d = \text{int}(var/N) \tag{14}$$

onde var_d é a variável após a discretização, var é variável original, N é um parâmetro inteiro que controla a redução do espaço de estados e int(.) é uma função que transforma um número real em um inteiro. Então se $N = 5$, qualquer y real entre 0 e 4 será representado por uma variável discreta $y_d = 0$, reduzindo de forma considerável o espaço de estados mesmo se y fosse uma variável inteira. A redução do espaço pode aumentar a capacidade de generalização e reduzir o tempo de treinamento do Q Learning pois quanto maior o espaço maior a probabilidade de existirem estados que ainda não foram visitados pelo agente e em estados não visitados o agente não possui nenhum tipo de informação, levando a uma má decisão, mesmo se o agente já tivesse vistado um estado muito semelhante. A discretização poderia ser usada nas outras metodologias mas só foi aplicada no Q Learning.

É possível também acrescentar variáveis que podem ter alguma importância para o agente, a representação de um estado não depende da velocidade do pássaro, contudo, o modelo de transição depende, podendo então ser uma variável importante. Variáveis como a distância vertical (d_y^{ci} e d_y^{cs}) e horizontal (d_x) entre o pássaro e algum dos canos também podem ser utilizadas, apesar não fazerem parte da representação de um estado são obtidas das variáveis fundamentais e possuem um significado importante. Na busca por formas de observação de estado eficientes, foram também pensadas em variáveis binárias que representam se o pássaro está acima ou abaixo de um dos canos, esse tipo de variável possui informação importante e pode substituir variáveis que possuem maior número de valores possíveis. A variável α é igual à 0 se o pássaro está abaixo de cano de baixo e 1 caso contrário e a variável β é igual à 0 se o pássaro está abaixo do cano de cima e 1 caso contrário.

Afim de identificar a melhor a representação possível, foram feitos testes empíricos que serão descritos na seção de resultados. As representações consideradas se encontram na Tabela 1. É importante destacar que as representações consideradas no Q Learning sempre utilizam variáveis já discretizadas.

Tabela 1: Parâmetros de Representação de Estados.

Estado	Vetor de variáveis observáveis
s_1	$(d_y^{ci}, d_y^{cs}, d_x, v)$
s_2	(y, y_{cs}, y_{ci})
s_3	$(y - y_{ci}, v)$
s_4	$(d_y^{ci}, v, \alpha, \beta)$

Treinamento

O treinamento no contexto do Flappy Bird consiste no agente jogar o jogo diversas vezes até o tempo total de treinamento ultrapassar o limite de 30 minutos ou até a pontuação média do agente ser superior a 100. O treinamento do agente utilizando Q Learning e PPO segue os passos básicos de um treinamento por aprendizado por reforço. O agente observa o estado e escolhe a ação seguindo uma política, é então fornecida uma recompensa ao agente e ação causa uma transição para um novo estado, concluindo um episódio. Através das recompensas obtidas nos episódios são aprendidas as políticas.

A definição de uma boa função de recompensa é fundamental para garantir que o agente aprenda a função desejada e é uma das fases mais complexas de um processo de aprendizado por reforço.

Algumas funções de recompensa para o Flappy Bird foram elaboradas e testadas. A função de recompensa mais simples e explícita seria

$$r_t^1 = \begin{cases} 1, \text{ se fez ponto} \\ 0, \text{ se não} \end{cases} \tag{15}$$

o problema dessa função é que a recompensa só será obtida se um conjunto de ações muito específico for feito ao acaso, é importante lembrar que no início do treinamento não há qualquer informação prévia sobre o problema. Essa função irá requerer um grande tempo de treinamento e não se quer a garantia de algum aprendizado. Pode-se então formular funções mais explícitas, na maioria dos trabalhos feitos sobre treinamentos de agente por RL para jogar Flappy Bird utilizam o seguinte formato de função:

$$r_t^2 = \begin{cases} a, \text{ se está vivo} \\ -b, \text{ se colidiu} \end{cases} \tag{16}$$

onde a e b são números reais constantes. A função r_t^2 foi utilizada nos testes preliminares e foi constatado que há aprendizado mas ocorre de for lenta. Foram elaboradas então funções ainda mais explícitas, guiando a o agente há uma estratégia que aparentemente é eficaz r_t^3, r_t^4 e r_t^5

$$r_t^3 = \begin{cases} a, \text{ se está vivo e entre os canos} \\ -b, \text{ se colidiu} \end{cases} \tag{17}$$

$$r_t^4 = \begin{cases} a, \text{ vivo, entre os canos e } d < \delta \\ -b, \text{ se colidiu} \end{cases} \tag{18}$$

$$r_t^5 = \begin{cases} a, \text{ pulou estando abaixo do centro} \\ a, \text{ não pulou estando acima do centro} \\ -b, \text{ se colidiu} \end{cases} \tag{19}$$

onde d é a distância vertical entre o pássaro e o centro dos tubos e δ é um número real que define o quão próximo do centro o pássaro deve estar do centro para ganhar a recompensa, em r_t^3 o centro se refere ao centro dos tubos. Para determinar a melhor função de recompensa foram feitos testes empíricos que serão descritos na seção de resultados. Diferentemente Q Learning e do PPO, que utilizam

uma função de recompensa para cada ação executada pelo agente, o Algoritmo Genético possui uma função de Fitness que avalia o quão boa é uma solução, dessa forma, foi utilizado a pontuação do agente durante o jogo como Fitness de cada um dos indivíduos, utilizando a recompensa r_t^1 e a representação de ambiente s_2

Resultados

NEAT

Considerando as estratégias apresentadas no presente trabalho, o Algoritmo Genético, base para o funcionamento do algoritmo NEAT, é o método que mais difere dos outros que serão apresentados aqui. Uma dessas diferenças é em relação à forma de avaliação da solução encontrada, bem como no método de busca em si. Todas os experimentos utilizando o algoritmo NEAT foram implementados com o uso da biblioteca NEAT-Python.

O NEAT utiliza a função Fitness de cada indivíduo para a sua avaliação, função que, no caso apresentado aqui, será atribuída com a pontuação obtida pelo agente durante o jogo, pontuação que é obtida apenas quando o agente perde, o que pode não acontecer com um agente treinado, sendo assim, o agente que atingiu 100 pontos durante os experimentos foi considerado treinado, tendo um limite de 100 gerações para o treinamento.

Durante os testes iniciais, as topologias mais simples atingiram pontuações mais altas no decorrer das gerações, porém com baixa consistência nos indivíduos de uma geração para outra, onde houveram casos em que uma geração convergia para um indivíduo treinado, e na próxima geração, nenhum indivíduo conseguia marcar pontos. Apenas de encontrar soluções com boas pontuações, a falta de consistência entre as gerações foi uma característica marcante durante a execução do treinamento utilizando o algoritmo NEAT.

O algoritmo NEAT possui configurações para a execução da busca usando Algoritmo Genético e para o cálculo de cada indivíduo, dessa forma, existem vários parâmetros para as duas funcionalidades, detalhados a seguir.

- num_inputs: Número de neurônios de entrada da rede neural;
- num_hidden: Número de neurônios da camada oculta da rede neural;
- $num\_outputs$: Número neurônios de saída da rede neural;
- $activation\_default$: Função de ativação dos neurônios;
- $weight\_mutate\_rate$: Taxa de mutação dos pesos de ligações entre os neurônios;
- $weight\_mutate\_power$: Amplitude de valores em que um novo peso mutado pode ser escolhido.

Considerando a falta de consistência entre as gerações e as possíveis combinações de parâmetros, tanto da execução do NEAT quando da própria rede neural que é representada por cada indivíduo dentro da população, foram criadas combinações utilizando os parâmetros mencionados acima, com excessão dos parâmetros num_inputs e $num\_outputs$, uma vez que esses são obtidos a partir das observações do ambiente e utilizados para decidir a ação executada pelo agente durante o jogo. Apesar da biblioteca NEAT-Python apresentar diversas outras opções de parâmetros, apenas esses foram variados devido o tempo necessário para avaliar cada cenário diferente.

Tabela 2: Configuração NEAT 1

Hyper parâmetro	Valor
activation_default	relu
num_hidden	0
weight_mutate_power	10

Tabela 3: Configuração NEAT 2

Hyper parâmetro	Valor
activation_default	relu
num_hidden	0
weight_mutate_power	1

Tabela 4: Configuração NEAT 3

Hyper parâmetro	Valor
activation_default	relu
num_hidden	0
weight_mutate_power	0.5

Foram realizados experimentos com outras configurações, aumentando a quantidade de neurônios da camada oculta, porém nesses experimentos nenhum indivíduo foi capaz de atingir a marca de 100 pontos. É possível observar um comportamento parcialmente aleatório na pontuação dos melhores indivíduos de cada geração, além da falta de consistência entre a pontuação de uma geração e a pontuação da geração posterior, como pode ser observado na Figura 9

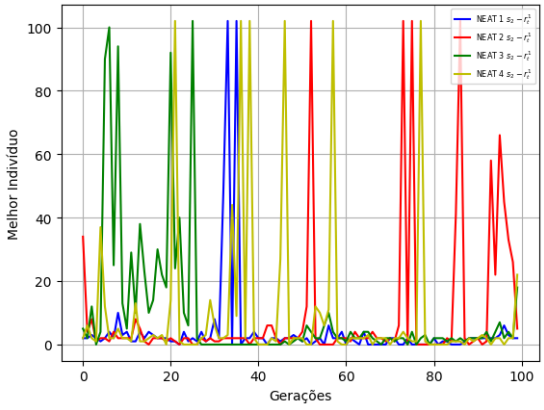

Figura 9: Pontuação do melhor indivíduo de cada geração ao longo das gerações

Tabela 5: Configuração NEAT 4

Hyper parâmetro	Valor
activation_default	relu
num_hidden	0
weight_mutate_power	0.1

Observando a pontuação média de cada geração na Figura 10, é possível afirmar que existe uma disparidade muito grande entre os indivíduos dentro de uma mesma geração, uma vez que a pontuação média máxima foi de 2 pontos, mesmo em gerações com indivíduos que conseguiram atingir o máximo de 100 pontos.

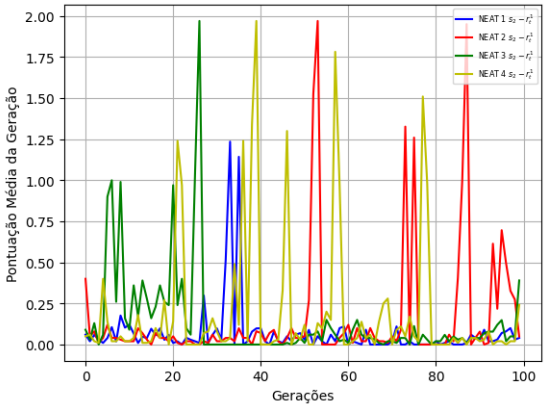

Figura 10: Pontuação média de cada geração

Uma das configurações possíveis no NEAT é a taxa de mutação, que pode ser aplicada ao peso das ligações, adição e remoção de neurônios, adição e remoção de ligações, o que inclui o elemento de aleatoriedade no processo de busca e pode explicar o comportamento parcialmente aleatório dos indivíduos no decorrer das gerações. Após um ajuste na taxa de mutação aplicada ao NEAT de 70%, padrão da biblioteca NEAT-python para 5% e 10%, o comportamento dos agentes no jogo deixou de ser variado e passou todos eles começaram a realizar as mesmas ações, como pode ser observado na Figura 11.

Apesar da existência de 100 indivíduos no experimento, apenas dois ficam visíveis pois estão se sobrepondo e realizando todos as mesmas ações, mesmo cada indivíduo representando uma rede neural artificial diferente.

Q Learning

Antes da geração de qualquer resultado, foram definidos valores base para os parâmetros de controle, se não forem explicitados os valores dos parâmetros pode-se considerar que os valores base foram utilizados. Os valores base se encontram na Tabela 6. A política escolhida no treinamento foi a ϵ-greedy, onde o parâmetro ϵ é definido como uma função do número de iterações (i), se $i < 1000$ então $\epsilon = \epsilon_0$, caso $i > 1000$ ϵ é decrementado em 0.01 sempre que o número de iterações é múltiplo de 500. Essa função garante uma maior

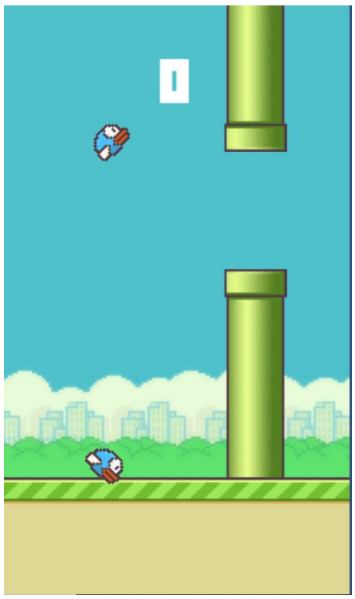

Figura 11: Captura da tela com os agentes realizando as mesmas ações

exploration no ínicio do treinamento e maior *exploitation* no final.

Tabela 6: Parâmetros base para o Q Learning.

Parâmetro	Valor
γ	0.9
μ	0.7
ϵ_0	0.5
N	5

Após alguns testes preliminares foi possível notar que a estratégia de treinamento padrão não era tão eficiente, principalmente considerando o Q Learning. A grande dificuldade da aplicação da metodologia tradicional está associada ao fato de não haver distinção entre passar um tubo no início do jogo ou no fim e a recompensa ir se acumulando ao longo do jogo. Essa indiferença pode levar a uma confusão na tabela com os valores de Q, impedindo uma maior progressão no desempenho do agente e até perda de desempenho após um certo número de iterações. Outro fator importante que torna o processo tradicional pouco eficiente é que o tempo do jogo aumenta com a performance do agente e consequentemente aumenta o tempo de treinamento, o tempo de jogo de um agente que consegue fazer 100 pontos é quase 100 vezes maior que o tempo de um jogo onde o agente consegue fazer 1. A Figura 12 ilustra as dificuldades descritas.

Essas dificuldades levaram a elaboração de uma nova metodologia de treino, ao invés de deixar o agente jogar livremente, o jogo se encerra quando o agente consegue 2 pontos. Como não há distinção entre a representação de um estado de baixa pontuação para um de alta pontuação e os canos são gerados de forma aleatória, um agente que consegue passar os dois primeiros canos de maneira muito con-

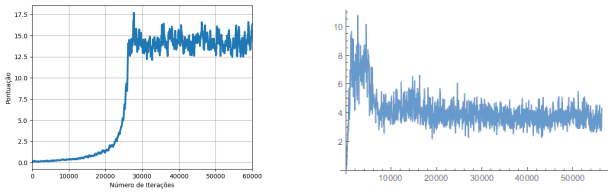

Figura 12: Pontuação em função do número de iterações para o caso onde o desempenho assintótico é limitado (esquerda) e caso onde o desempenho cai após um certo número de iterações (direita).

sistente também irá conseguir passar os demais com alta probabilidade e consequentemente terá um bom desempenho no jogo. Essa metodologia também impõe um tempo máximo para um jogo, tornando o processo de aprendizado muito mais rápido. Aplicando esse processo considerando os parâmetros base, representação de estado s_1 e função de recompensa r_4 foi obtida uma pontuação média de 42.5 e a melhor pontuação obtida foi de 279 pontos, considerando 1 hora de jogo de um agente treinado. Note que a média de 42.5 é bem superior a performance assintótica obtida pela metodologia tradicional.

Após a mudança da metodologia de treinamento foram definidas a melhor função de recompensa e a melhor representação de estados. O estado foi fixado e diferentes funções de recompensas desenvolvidas foram avaliadas, esse resultado se encontra na Figura 13.

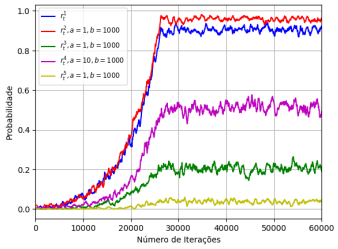

Figura 13: Probabilidade de concluir a tarefa em função do número de iterações considerando diferentes funções de avaliação.

Claramente o melhor desempenho ocorre utilizando a função de recompensa r_4, sendo então adotada nos demais testes. A função de recompensa foi então fixada e as diferentes representações foram consideraras, esse resultado se encontra na Figura 14 . Mais uma vez há uma representação bem superior as demais, s_1, esta representação foi adotada nos demais testes realizados.

Após a definição de parâmetros base, metodologia de treinamento, função de recompensa e representação de estado, foram feitos testes para identificar o melhor conjunto de parâmetros e entender como esses parâmetros impactam no aprendizado. Nesses testes é variado apenas o parâmetro que se deseja estudar enquanto os demais são fixados aos valores base. Na Figura 15 se encontra o teste para o parâmetro ϵ_0, é importante deixar claro que os gráficos desta seção consideram uma média de 5000 jogos realizados, sem a aplicação da média os resultados apresentariam muitas oscilações o que dificulta suas interpretações.

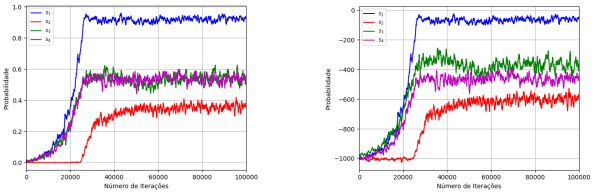

Figura 14: Probabilidade de concluir a tarefa (esquerda) e recompensa acumulada (direita) em função do número de iterações considerando diferentes modelos de observação.

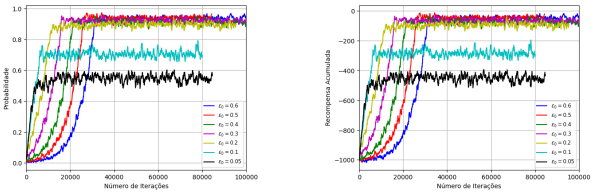

Figura 15: Probabilidade de concluir a tarefa (esquerda) e recompensa acumulada (direita) em função do número de iterações considerando múltiplos valores de ϵ_0.

O parâmetro ϵ_0 apresentou um grande impacto no desempenho do agente tanto na consistência em realizar tarefa quanto na recompensa acumulada obtida. Para ϵ_0 de 0.6 até 0.3 é possível observar que um aumento de ϵ_0 contribui em maior velocidade de aprendizado nas iterações inicias mas um mesmo desempenho assintótico, no entanto, para $\epsilon_0 < 0.3$ apesar de se manter a maior velocidade inicial o desempenho assintótico é inversamente proporcional à ϵ_0. Esse resultado evidencia a necessidade de uma maior *exploration* inicial para se aprender uma solução mais eficiente no futuro.

Foram testados também os parâmetros γ e μ e os resultados estão nas Figuras 16 e 17. Esses parâmetros não apresentaram grande influência no desempenho, tanto do ponto de vista da velocidade inicial de aprendizado quanto do desempenho assintótico.

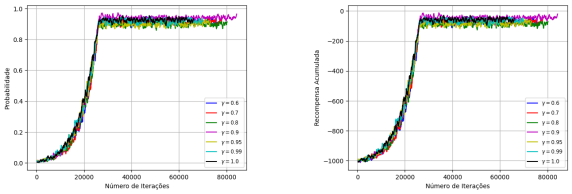

Figura 16: Probabilidade de concluir a tarefa (esquerda) e recompensa acumulada (direita) em função do número de iterações considerando multiplos valores de γ.

O parâmetro de controle da discretização N também foi avaliado, os resultados estão na Figura 18 . Há um notável

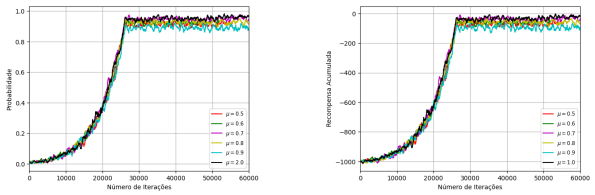

Figura 17: Probabilidade de concluir a tarefa (esquerda) e recompensa acumulada (direita) em função do número de iterações considerando multiplos valores de μ

impacto no desempenho assintótico e o melhor desempenho ocorre para $N = 6$. Um N muito pequeno diminui a capacidade de generalização do agente enquanto que um N muito grande pode simplificar demais o espaço de estados onde informações importantes podem ser perdidas.

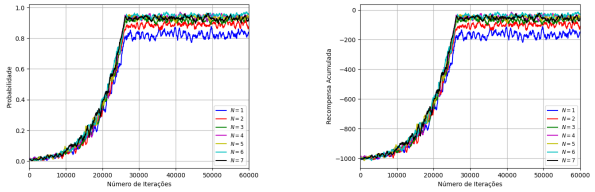

Figura 18: Probabilidade de concluir a tarefa (esquerda) e recompensa acumulada (direita) em função do número de iterações considerando multiplos valores de N.

Os parâmetro ótimos identificados se encontram na Tabela 7, com esses parâmetros o agente treinado obteve pontuação média de 49.18 e melhor pontuação de 290 pontos considerando 1 hora de jogo, o agente já treinado utiliza a política gulosa e escolhe sempre a ação que maximiza Q.

Tabela 7: Parâmetros ótimos encontrados para o Q Learning.

Parâmetro	Valor
γ	0.9
μ	0.7
ϵ_0	0.5
N	6

PPO

O PPO permite que múltiplos agentes treinem de forma simultânea para agilizar o processo de treinamento, dessa forma o ambiente criado para o treinamento possui 30 agentes que treinam de forma simultânea. Por também ser uma estratégia de aprendizado por reforço, o processo de treinamento de PPO ocorre de forma muito semelhante ao do Q Learning, essa característica resultou na reprodução dos mesmos problemas encontrados na estratégia de treinamento padrão do Q Learning. Sendo assim, a estratégia de treinamento do PPO também foi adaptada para a estratégia de treinamento onde o jogo se encerra quando o agente consegue 2 pontos, e foi mantida a configuração de 30 agentes no ambiente de treinamento.

Assim como no Q Leaning foram realizados testes preliminares, utilizando valores bases para os hiper parâmetros e a nova estratégia de treinamento, o objetivo dos testes era identificar uma configuração ótima de representação de estado e função de recompensa para serem utilizado no aprendizado com o algoritmo PPO. Os valores bases podem ser observados nas Tabelas 8, 9 e 10.

Tabela 8: Hiper parâmetros do PPO.

Hyper parâmetro	Valor
batch_size	128
buffer_size	2048
learning_rate	3.0e-4
beta	5.0e-3
epsilon	0.2
lambd	0.95
num_epoch	3

Tabela 9: Configurações da rede neural do PPO.

Hyper parâmetro	Valor
normalize	false
num_layers	2
hidden_units	64

Tabela 10: Configurações de sinais de recompensa do PPO.

Hyper parâmetro	Valor
gamma	0.99
strength	1.0

Abaixo segue uma descrição de cada parâmetros mencionado nas tabelas 8, 9 e 10.

1. $batch\_size$ Número de experiências em cada iteração de descida gradiente. O valor deste parâmetro deve ser sempre várias vezes menor que $buffer\_size$. Ao utilizar um espaço de ação contínua, esse valor deve ser grande. Já em um espaço de ações discreta, este valor deve ser menor.
2. $buffer\_size$ Número de experiências que devem ser coletadas antes de atualizar o modelo de política. Normalmente, um valor maior corresponde a atualizações de treinamento mais estáveis.
3. $learning\_rate$ Taxa de aprendizagem inicial para descida gradiente. Corresponde à força de cada etapa de atualização da descida do gradiente.
4. $beta$ Força da regularização da entropia, alterar os valores deste parâmetro pode tornar a política ”mais aleatória”, garantindo que os agentes explorem adequadamente o espaço de ação durante o treinamento. Este parâmetro deve ser ajustado de forma que a entropia diminua lentamente, a medida que ocorrem aumentos na recompensa. Se a entropia cair muito rapidamente, é recomendado o aumento do valor deste parâmetro. Caso a entropia caia muito lentamente, é recomendado a diminuição do valor.

5. $epsilon$ Influencia a velocidade com que a política pode evoluir durante o treinamento. Corresponde ao limite aceitável de divergência entre as políticas antigas e novas durante a atualização do gradiente descendente. Definir um valor pequeno, resultará em atualizações mais estáveis ao custo de tornar o processo de treinamento mais lento.
6. $lambd$ Parâmetro de regularização $lambda$ usado ao calcular a estimativa de vantagem generalizada (GAE). Isso pode ser considerado como o quanto o agente depende de sua estimativa de valor atual, para calcular uma estimativa de valor atualizada. Valores baixos correspondem a confiar mais na estimativa do valor atual podendo resultar em um alto viés. Por outro lado, valores altos correspondem a confiar mais nas recompensas reais recebidas no ambiente, oque pode resultar em muitas variações.
7. num_epoch Número de passagens a serem feitas no buffer de experiência ao realizar a otimização de gradiente descendente. Diminuir o valor deste parâmetro garantirá atualizações mais estáveis, ao custo de um aprendizado mais lento.
8. $normalize$ Se a normalização é aplicada às entradas de observação vetorial. Essa normalização é baseada na média e na variância da observação do vetor. A normalização pode ser útil em casos com problemas complexos de controle contínuo, mas pode ser prejudicial com problemas de controle discreto mais simples.
9. num_layers O número de camadas ocultas na rede neural. Corresponde a quantas camadas ocultas estão presentes após a entrada de observação. Para problemas simples, um valor menor de camadas provavelmente serão treinadas com mais rapidez e eficiência. Um número maior de camadas podem ser necessárias para problemas de controle mais complexos.
10. $hidden\_units$ Número de unidades nas camadas ocultas da rede neural. Corresponde a quantas unidades estão em cada camada conectada da rede neural. Para problemas simples em que a ação correta é uma combinação direta das entradas de observação, valores pequenos podem obter melhores resultados. Para problemas em que a ação é uma interação muito complexa entre as variáveis de observação, valores altos podem obter melhores resultados.
11. $gamma$ Fator de desconto para recompensas futuras provenientes do ambiente. Isso pode ser considerado como o quanto no futuro o agente deve se preocupar com as possíveis recompensas. Em situações em que o agente deve estar agindo no presente a fim de se preparar para recompensas no futuro distante, esse valor pode ser alto. Nos casos em que as recompensas são mais imediatas, os valores podem ser menores. Nos 2 casos, o valor deve ser estritamente menor que 1.
12. $strength$ Fator pelo qual se multiplica a recompensa dada pelo ambiente. Os intervalos típicos variam dependendo do sinal de recompensa.

Após os teste preliminares e a identificação da representação de estado e função de recompensa, se iniciaram os testes para verificar quais hiper parâmetros seriam modificados com o objetivo de atingir resultados melhores no treinamento. Os testes foram feitos através da utilização do valor base para a maior parte dos hiper parâmetros, e a cada nova execução dos testes eram realizadas alterações em somente 1 ou 2 valores dos hiper parâmetros. Analisando os resultados, foram selecionados os hiper parâmetros num_layers, $hidden\_units$ e $beta$.

A escolha dos parâmetros num_layers e $hidden\_units$ teve como objetivo utilizar RNAs com diferentes complexidades em relação a estrutura interna da rede. Já o parâmetro $beta$ teve o objetivo de fazer o algoritmo explorar os estados de forma mais adequada, fazendo mais testes antes de diminuir a entropia isso poderia resultar em uma generalização melhor ao final do treinamento. Esses parâmetros foram utilizados em um processo de testes mais elaborado, o objetivo desse teste foi encontrar uma configuração ótima considerando alterações nesses 3 hiper parâmetros. Cada execução de testes utilizada a função de recompensa r_t^5 e passa por 1 milhão de interações e levando ao menos 30 minutos para serem executadas. Nas tabelas 11, 12, 13 e 14 é apresentada 4 configurações utilizadas para treinar os agentes e obter os resultados apresentados nas figuras 19, 20 e 21.

Tabela 11: Hiper parâmetros do Cenário 1.

Hyper parâmetro	Valor
num_layers	2
hidden_units	64
beta	5.0e-3

Tabela 12: Hiper parâmetros do Cenário 2.

Hyper parâmetro	Valor
num_layers	1
hidden_units	32
beta	5.0e-3

Tabela 13: Hiper parâmetros do Cenário 3.

Hyper parâmetro	Valor
num_layers	2
hidden_units	64
beta	4e-1

Após a análise dos testes, foi observado que a melhor configuração de parâmetro foi o $Cenário1$, por ter conseguido treinar o agente de forma mais rápida, e mantendo uma boa generalização do agente quando comparado aos outros. As outras configurações obtiveram resultados finais próximos do $Cenário1$, mas levaram mais tempo para treinar o agente. Dessa forma, os parâmetros da $Cenário1$ foram selecionados e utilizados nos novos testes. Após ter encontrado as configurações ótimas para os hiper parâmetros, foi iniciada uma nova etapa de teste onde o cada execução passava pro 2 milhões de interações e demorava ao menos 1 hora para serem executadas. O teste consistia em utilizar

Tabela 14: Hiper parâmetros do Cenário 4.

Hyper parâmetro	Valor
num_layers	1
hidden_units	32
beta	4e-1

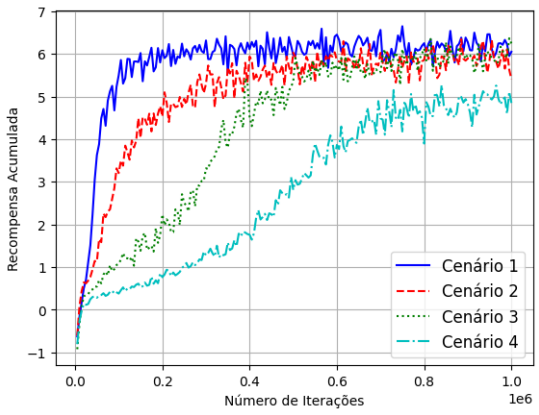

Figura 19: Recompensa acumulada em função do número de interações.

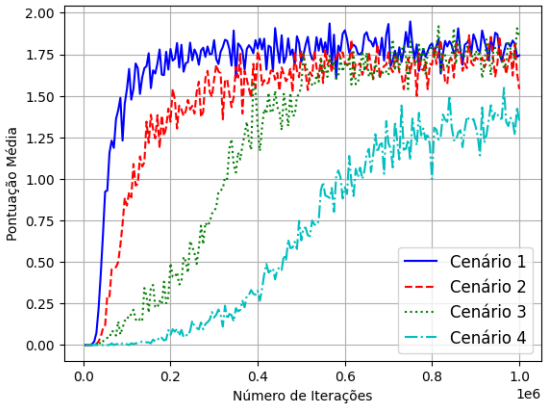

Figura 20: Pontuação média em função do número de interações.

as representações de estados s_1, s_2 e s_3 mencionadas anteriormente e analisar o desempenho de cada uma dessas representações ao combiná-las com as funções de recompensa: r_t^1, r_t^4 e r_t^5. A figuras 22 a 27 apresentam os resultados deste teste.

Analisando as figuras mencionadas anteriormente, foi possível constatar que dependendo da função de recompensa a melhor representação de estado também irá variar. A figura 28 apresenta as informações de pontuação média para cada treinamento realizado, através desta figura é possível perceber que a combinação que obteve o melhor resultado foi s_3 com a recompensa r_t^5, essa combinação foi capaz de treinar o agente de forma mais rápida, e manteve uma boa generalização do agente quando comparado aos outros.

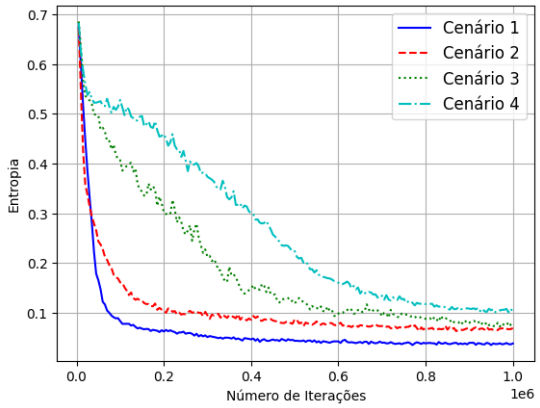

Figura 21: Variação da entropia em função do número de interações.

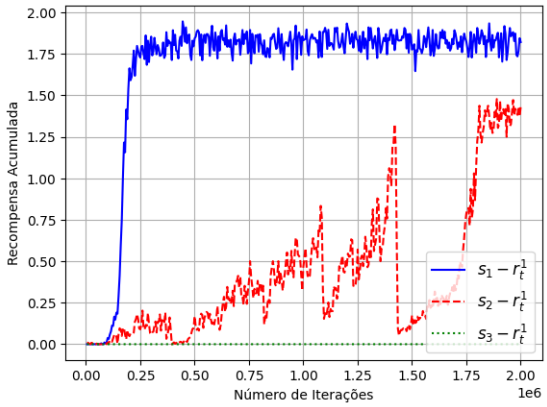

Figura 22: Recompensa acumulada em função do número de interações. Cada curva representa uma representação de estado s_n diferente, utilizando a função de recompensa r_t^1.

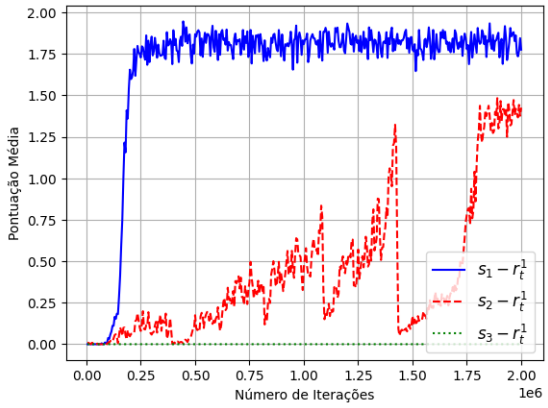

Figura 23: Pontuação média em função do número de interações. Cada curva representa uma representação de estado s_n diferente, utilizando a função de recompensa r_t^1.

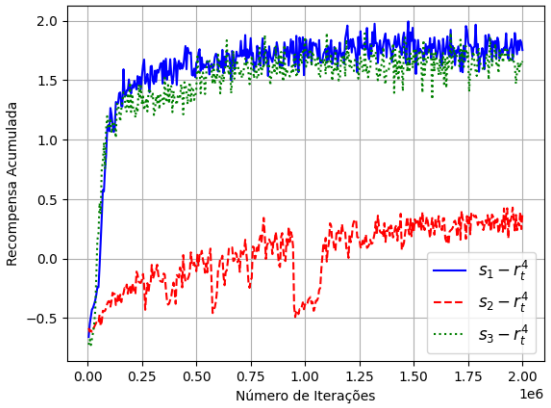

Figura 24: Recompensa acumulada em função do número de interações. Cada curva representa uma representação de estado s_n diferente, utilizando a função de recompensa r_t^4.

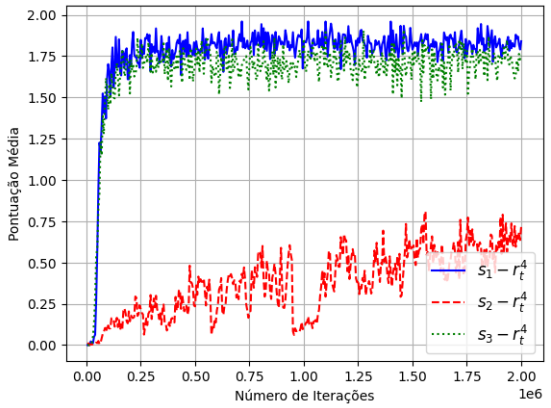

Figura 25: Pontuação média em função do número de interações. Cada curva representa uma representação de estado s_n diferente, utilizando a função de recompensa r_t^4.

No final de cada teste o algoritmo PPO da Unity, gera um arquivo de extensão $.nn$ que contém o modelo gerado durante o treinamento. Esse arquivo pode ser utilizado para fazer com que um novo agente use oque foi aprendido durante o processo de teste. Para avaliar como o novo agente irá se sair, foi utilizado o arquivo gerado pela representação de estado s_3 com a recompensa r_t^5 por ter obtido os melhores resultados. Ao utilizar esse modelo no jogo normal, sem nenhuma restrição de pontuação, após 30 minutos de jogo obtivemos uma pontuação média de 1.5 e máxima de 10. Este resultado é um tanto curioso, uma vez que os agentes conseguiram se sair bem durante os treinamentos era esperado que o modelo gerado se saísse melhor. Desse modo, o arquivo $.nn$ que contêm o modelo, parece apresentar algum tipo de problema ao reproduzir o conhecimento adquirido nos testes.

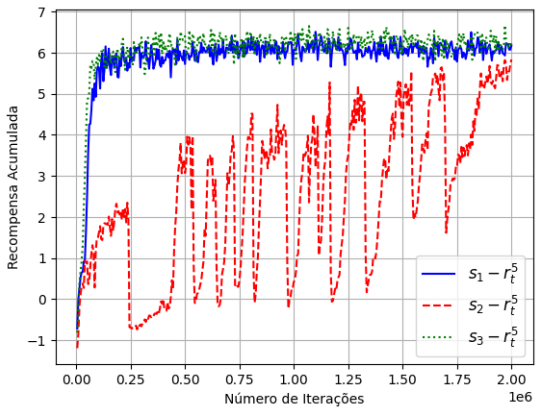

Figura 26: Recompensa acumulada em função do número de interações. Cada curva representa uma representação de estado s_n diferente, utilizando a função de recompensa r_t^5.

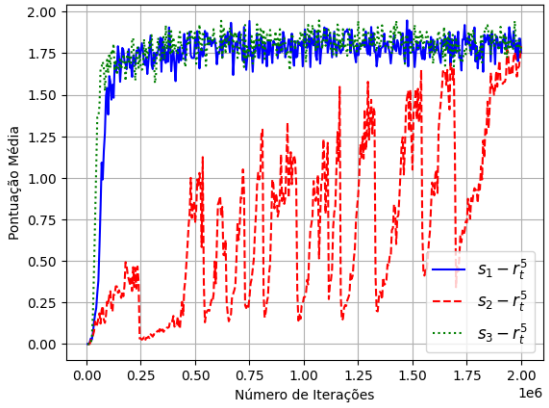

Figura 27: Pontuação média em função do número de interações. Cada curva representa uma representação de estado s_n diferente, utilizando a função de recompensa r_t^5.

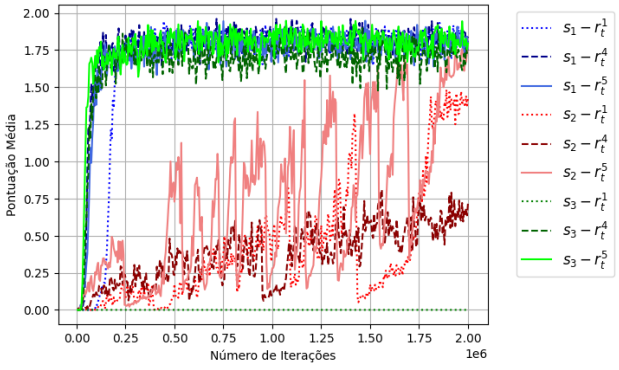

Figura 28: Pontuação média em função do número de interações. Cada curva representa uma representação de estado s_n diferente, utilizando as funções de recompensa r_t^y.

Conclusões

O jogo Flappy Bird foi formulado como um problema que pode ser resolvido através de uma metodologia de inteligência artificial, onde foi proposto um modelo de representação de estado e um modelo de transição onde o objetivo do agente é nunca perder o jogo ou conseguir uma boa performance de forma consistente. Foram também desenvolvidas funções de recompensa e de fitness para guiar o agente à encontrar uma boa solução para o problema. A forma como o agente percebe o ambiente tem grande impacto em seu desempenho, foram consideradas diferentes variáveis para compor as observações dos agentes e diferentes resultados foram obitdos.

O algoritmo NEAT apresentou problemas de consistência nos resultados obtidos durante o treinamento e no decorrer das gerações, apresentando pontuações no limite estabelecido nos experimentos utilizando AG em uma geração, porém zerando a pontuação na próxima geração, problema causado pelo operador genético de mutação implementado pelo algoritmo. Simulações com taxas de mutação de 5% a 10% o agente não obteve resultados satisfatórios, não conseguindo evoluir ao longo das gerações, inclusive, fazendo com que todos os agentes presentes na simulação executassem exatamente as mesmas ações, mesmo com cada agente sendo controlado por uma rede neural diferente, causando assim perca da variabilidade das soluções obtidas. Outra característica apresentada por essa solução é que o agente é evoluído apenas quando finalizada a geração completa, o que pode causar estagnação nos indivíduos obtidos e que possuem desempenho razoável, não evoluindo seu desempenho até que o mesmo finalize a sua execução atingindo um dos canos.

O algoritmo Q Learning foi avaliado considerando sempre representações discretizadas das variáveis, sem a discretização a performance do algoritmo é prejudicada e ao mesmo tempo a quantidade de memória demandada torna a utilização de variáveis contínuas ou que possuem uma grande faixa de valores inviável. Foi desenvolvida uma metodologia diferente de treinamento onde é apenas considerada a tarefa de passar por dois canos, e o aprendizado obtido nessa tarefa é transferido para o problema de passar pelo maior número de canos. Essa metodologia obteve um melhor resultado, um agente treinado na nova metodologia obteve uma pontuação média duas vezes maior que um agente treinado na metodologia antiga e em um intervalo de tempo menor. Diferentes funções de recompensa foram testadas assim como diferentes formas de observação, as que obtiveram o melhor resultado foram r_4 e s_1. Os parâmetros de controle também foram estudados, o parâmetro ϵ_0 se mostrou um parâmetro muito importante, a *exploration* inicial é fundamental para o desempenho futuro do agente, o parâmetro N se mostrou importante, a forma de discretizar o problema está diretamente relacionada à capacidade de generalização do agente. Uma vez identificados os parâmetros ótimos, um teste final foi realizado, o agente treinado obteve pontuação média de 49.18 e maior pontuação de 290, considerando 30 minutos na fase de treino e 1 hora de jogo real.

O algoritmo PPO apresentou um bom desempenho durante o treinamento, ele conseguiu convergir rapidamente para um estado onde a maior parte dos agentes envolvidos no treinamento estavam obtendo a pontuação máxima. Foi utilizada a mesma metodologia de treinamento do Q Learning, porém com a vantagem de ser possível utilizar diversos agentes no treinamento deixando este processo mais rápido. Este algoritomo se mostrou flexível ao conseguir trabalhar com dados contínuos e discretos sem necessitar de uma grande quantidade de memória, além de ter obtido bons resultados no treinamento utilizando diferentes representações de estados e funções de recompensa. Apesar da flexibilidade, ainda é bastante importante se atentar na tupla que representa a combinação de representação de estados e função de recompensa, pois dependendo dessa combinação o algoritmo pode não convergir para um bom resultado. Foram encontrados problemas na utilização do modelo gerado após o treinamento, aparentemente o modelo gerado não conseguiu reproduzir os mesmo resultados obtidos no processo de treinamento, durante o treinamento foi possível observar que a maior parte dos agentes atingiam a pontuação máxima mas quando foi utilizado o modelo gerado no jogo normal, sem nenhuma restrição de pontuação, após 30 minutos de jogo obtivemos uma pontuação média de 1.5 e máxima de 10 que é um resultado abaixo do esperado considerando o desempenho dos agentes nos teste.

Todos os algoritmos conseguiram realizar a tarefa de jogar o jogo e obter uma alta pontuação, alguns de forma mais consistente que os outros. Considerando os experimentos feitos com alguns voluntários, um humano que nunca jogou o jogo consegue uma pontuação média de 4 pontos em uma hora de jogo, obtendo 21 no melhor dos casos. Os agentes tiveram um desempenho superior no mesmo tempo de treinamento. O algoritmo Q Learning se mostrou o mais consistente para resolver o Flappy Bird, obtendo a maior pontuação média. No entanto, em problemas mais complexos onde o espaço de estados e ações é contínuo ou muito grande não é possível a aplicação do Q Learning.

Participação de cada componente

O trabalho foi dividido em 5 fases principais: Escolha e definições de característica dos ambientes e algoritmos, implementação do jogo na Unity e Pygame, implementação dos algoritmos e ambiente de treinamento utilizados, coleta de resultados e redação do relatório final. Todos os componentes participaram ativamente de todas as fases do trabalho. Pode-se destacar a participação do Rodrigo na implementação do jogo em Pygame e do Leonardo na implementação do jogo na Unity. Além disso cada integrante ficou responsável por implementar e coletar os resultados de um dos algoritmos utilizados, o Rodrigo ficou responsável pelo Q Leaning, o José responsável pelo algoritmo Genético e o Leonardo pelo PPO. A redação do relatório final foi realizada em conjunto por todos os componentes, sendo que o detalhamento específico de cada algoritmo ficou sendo de responsabilidade do componente que realizou a implementação.

Referências

AlphaGo versus Lee Sedol. Disponível em: https://en.wikipedia.org/wiki/AlphaGo_versus_Lee_Sedol. Acesso em: 12 Dezembro 2020.

Alves, J. d. S.; de Carvalho, C. L.; and de A Sabino, G. 2020. Aplicação de redes neurais artificiais em problemas de satisfação de restrições. *Revista de Sistemas e Computação-RSC* 9(2).

Faceli, K.; Lorena, A. C.; Gama, J.; and Carvalho, A. C. P. d. L. F. d. 2011. *Inteligência artificial: uma abordagem de aprendizado de máquina*. LTC.

França, I. 2019. Learning how to play bomberman with deep reinforcement and imitation learning. *Springer International Publishing* 121–133.

Furtado, M. I. V., and Furtado, R. C. 2020. Estudo avaliativo da emissão de co2 a partir de combustíveis fósseis utilizando redes neurais. *Conhecimento & Diversidade* 11(25).

Gonçalves, A. R. 2019. Algoritmos genéticos.

Goodfellow, I.; Bengio, Y.; and Courville, A. 2016. *Deep Learning*. The MIT Press.

Haykin, S. 2007. *Redes neurais: princípios e prática*. Bookman Editora.

LACERDA, E. d.; CARVALHO, A. d.; GALVÃO, C.; VALENÇA, M.; VIEIRA, V.; DINIZ, L.; and LUDERMIR, T. 1999. Sistemas inteligentes: aplicações a recursos hídricos e ciências ambientais. *Porto Alegre, RS: Universidade/UFRGS* 99–150.

Lanham, M. 2018. *Learn Unity ML-Agents – Fundamentals of Unity Machine Learning: Incorporate new powerful ML algorithms such as Deep Reinforcement Learning for games*. Packt Publishing.

Li, Y. 2018. Deep reinforcement learning.

Lopes, H. 2006. Fundamentos da computação evolucionária e aplicações. *Paraná: Bandeirantes*.

Marsland, S. 2009. *Machine Learning - An Algorithmic Perspective.* Chapman and Hall / CRC machine learning and pattern recognition series. CRC Press.

Pecenin, M. 2019. Otimização de escalonamento halide através de aprendizado por reforço. *SBC* 37–48.

Prudêncio, R. B. C. 2001. Projeto híbrido de redes neurais. Master's thesis, Universidade Federal de Pernambuco.

Pygame. Disponível em: https://www.pygame.org/. Acesso em: 12 novembro 2020.

Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; and Klimov, O. 2017. Proximal policy optimization algorithms.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; van den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; Dieleman, S.; Grewe, D.; Nham, J.; Kalchbrenner, N.; Sutskever, I.; Lillicrap, T.; Leach, M.; Kavukcuoglu, K.; Graepel, T.; and Hassabis, D. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529(7587).

Soder, A., and Malheiros, M. d. 2019. Avaliando a técnica de aprendizado por reforço neat quando aplicada a uma rede neural jogando um videogame de console de 8 bits. *Revista Destaques Acadêmicos* 11(4).

Stanley, K. O., and Miikkulainen, R. 2002. Evolving neural networks through augmenting topologies. *Evolutionary computation* 10(2):99–127.

Tsitsiklis, N, J. 1994. Asynchronous stochastic approximation and qlearning. *Machine Learning* 16(3):185–202.

Unity. Disponível em: https://unity.com/. Acesso em: 12 novembro 2020.

YAMAZAKI, A. 2004. Uma metodologia para otimização de arquiteturas e pesos de redes neurais.