UNIVERSIDADE FEDERAL DE SANTA CATARINA
CAMPUS FLORIANÓPOLIS

Robson Zagre Júnior

KITSUNE AI: UMA ARQUITETURA UTILIZANDO AGENTE BDI E APRENDIZAGEM POR REFORÇO PARA JOGAR JOGOS DE NES.

FLORIANÓPOLIS
2022

ROBSON ZAGRE JÚNIOR

KITSUNE AI: UMA ARQUITETURA UTILIZANDO AGENTE BDI E APRENDIZAGEM POR REFORÇO PARA JOGAR JOGOS DE NES.

Trabalho de Conclusão de Curso submetido à Universidade Federal de Santa Catarina, como requisito necessário para obtenção do grau de Bacharel em Ciências da Computação com a orientação do Prof. Dr. Maicon Rafael Zatelli

Florianópolis, 2022

UNIVERSIDADE FEDERAL DE SANTA CATARINA

ROBSON ZAGRE JÚNIOR

Esta Monografia foi julgada adequada para a obtenção do título de Bacharel em Ciências da Computação, sendo aprovada em sua forma final pela banca examinadora:

Orientador: Prof. Dr. Maicon Rafael Zatelli
Universidade Federal de Santa Catarina - UFSC

Profa. Dra. Jerusa Marchi
Universidade Federal de Santa Catarina - UFSC

Prof. Dr. Mauro Roisenberg
Universidade Federal de Santa Catarina - UFSC

Florianópolis, 5 de Agosto de 2022

Este trabalho é dedicado às crianças adultas que,
quando pequenas, sonharam em se tornar cientistas.
Fico feliz por ser uma delas ...

Resumo

A utilização de técnicas de Aprendizado por Reforço em Agentes demonstra uma união de diferentes frentes da Inteligência Artificial, necessitando-se de um bom modelo a fim de coordenar as abordagens escolhidas. Esse trabalho cria uma arquitetura modular que permite o desenvolvimento de um Agente que detecte objetos na tela para serem analisados pela abordagem *Belief–Desire–Intention (BDI)* amparada com algoritmos de Aprendizado por Reforço na concepção de planos com o objetivo de jogar jogos do NES, nesse caso *Super Mario Bros*. Na implementação, o agente é capaz de detectar obstáculos, inimigos e outros objetos da tela utilizados na criação de percepções. Essas formam o estado atual do jogo a fim de se descobrir a melhor ação possível, permitindo uma programação de mais alto nível atrelado a resultados mais específicos obtidos pelo treinamento. Foram utilizados os algoritmos QLearning e SARSA, demonstrando a arquitetura modular que permite experimentos e melhorias direcionados.

Palavras-chave: Aprendizado por Reforço, QLearning, SARSA, Agentes, BDI, Inteligência Artificial, Jogos Eletrônicos, NES.

Lista de ilustrações

Lista de tabelas

Lista de códigos

Lista de abreviaturas e siglas

Inteligência Artifial (IA)

Belief–Desire–Intention (BDI)

Reinforcement Learning (RL)

Nintendo Entertainment System (NES)

Machine Learning (ML)

Lei Geral de Proteção de Dados Pessoais(LGPD)

Natural Language Processing (NLP)

Convolutional Neural Networks (CNN)

Multi-Agent Programming Contest (MAPC)

Frames Per Second (FPS)

Read-Only Memory (ROM)

Application Programming Interface(API)

Sumário

1 Introdução

Como demonstrado por TURING (1950), uma possível avaliação da "inteligência" de uma máquina pode ser feita por meio de sua semelhança com o ser humano. Por mais que haja questionamentos sobre esse ponto (PAPADIMITRIOU, 2004), essa indagação proposta por Turing demonstra que quando pensamos em Inteligência Artificial (IA), a capacidade de adaptação à novas situações bem como a capacidade de raciocínio podem ser fundamentais para a identificação de uma entidade inteligente.

Com o constante avanço nas diferentes partes da IA, algoritmos de aprendizado de máquinas e de agentes estão sendo aprimorados, permitindo uma melhoria nos resultados (KRIZHEVSKY; SUTSKEVER; HINTON, 2012). Por mais que ambas as abordagens estejam voltadas para a solução de problemas aparentemente antagônicos, a utilização delas em conjunto pode ser benéfica para determinados problemas, como a execução da melhor ação por um robô jogando futebol (Zatelli et al., 2012).

Algoritmos inteligentes capazes de tomar decisões em um ambiente de forma autônoma, levando em consideração seus conhecimentos e raciocínios, são chamado de Agentes. O modelo *Belief–Desire–Intention* (BDI), proposto por Bratman (1987), é um exemplo de agente no qual, por meio de um ciclo de raciocínio, há o desenvolvimento de planos com o objetivo de alcançar uma meta definida. Por outro lado, possuímos algoritmos que detectam padrões e informações em dados observados, conseguindo prever um possível resultado no futuro com base em modelos estatísticos e funções matemáticas, chamados de Aprendizado de Máquinas. A técnica de Relearning, por exemplo, demonstra a capacidade de adaptabilidade e aprendizagem de acordo com os dados (BARTO, 1998).

A utilização de ambas as abordagens para a solução de uma problemática permite o desenvolvimento de um agente capaz de aprender com o meio em que está inserido, podendo obter respostas para ações que visam concretizar um plano. Essa abordagem permite também uma arquitetura mais flexível como vista por Bosello e Ricci (2020) onde a metodologia BDI fica responsável pela elaboração e execução de planos de alto nível, nesse caso achar o melhor caminho até o final do percurso, e o algoritmo de Aprendizado por Reforço, do inglês *Reinforcement Learning* (RL), fica responsável por desenvolver a melhor forma de navegação com base nas adversidades encontradas pelo caminho.

Um ambiente de jogos eletrônicos é um espaço virtual que apresenta, normalmente, um objeto controlado pelo jogador que efetua alguma ação com o intuito de cumprir um objetivo. Isso cria um cenário que permite testar ferramentas e estratégias de implementações de agentes autônomos capazes de se adaptar e aprender com o meio, além de facilitar a comparação (KRUKOSKI, 2019). Esse trabalho analisa a união de Aprendizado por

Reforço em Agentes BDI, utilizando o ambiente virtual do jogo *Super Mario Bros* pra avaliar a implementação de uma arquitetura de inteligência artificial híbrida capaz de jogar jogos de *Nintendo Entertainment System* (NES).

1.1 Objetivo

Desenvolvimento de uma arquitetura modular e escalável que permita a integração de aprendizagem por reforço à agente BDI com o intuito de jogar *Super Mario Bros* do NES.

1.1.1 Objetivos Específicos

Visando a concretização do objetivo geral, serão necessários os seguintes objetivos específicos:

- Pesquisar sobre algoritmos de RL, arquiteturas e modelagens de agentes BDI.
- Levantamento do estado da arte em IA híbrida.
- Elaboração de modelo e arquitetura geral do projeto.
- Implementação de agente BDI com RL no ambiente virtual.
- Avaliação da posposta e dos resultados.

1.2 Escopo do Trabalho

O presente trabalho desenvolve uma arquitetura que utiliza agente BDI e algoritmos de RL. Apresenta os conceitos e ferramentas necessários para seu desenvolvimento, bem como uma implementação de sua utilização voltado para o jogo *Super Mario Bros* utilizando os algoritmos QLearning e SARSA.

1.3 Organização

O presente trabalho está organizado da seguinte forma:

Capítulo 2: fundamentação teórica sobre Agentes e Aprendizado de Máquinas bem como seus focos de utilização e desafios.

Capítulo 3: apontamento dos trabalhos correlacionados visando entender as tecnologias utilizadas e a modelagem do problema.

Capítulo 4: demonstração da modelagem, arquitetura, ferramentas e desenvolvimento desse projeto.

Capítulo 5: resultados, comentários e discussões a respeito da arquitetura.

Capítulo 6: conclusão e cometários a respeito de trabalhos futuros.

2 Fundamentação Teórica

Nesse capítulo, é abordado conceitos necessários para a compreensão do trabalho. Primeiramente, na seção 2.1 é demonstrado questões simples e iniciais sobre Inteligência Artificial Híbrida, seguido do entendimento de Agentes inteligentes e da abordagem BDI na seção 2.2. Após isso, uma explicação sobre técnicas e alguns algoritmos de Aprendizado de Máquinas e Aprendizado por Reforço são introduzidos na seção 2.3, finalizando na seção 2.4 com uma explicação inicial de técnicas de visão computacional.

2.1 Inteligência Artificial Híbrida

A área de Inteligência Artificial pode ser dividia em: IA simbólica e IA não simbólica, cada uma possuindo assim suas particularidades quanto a implementação e solução de problemas (NILSSON, 2010). Essas áreas possuem subdivisões internas relacionadas às especificações encontradas no desenvolvimento, tecnologia, problemática e afins. A utilização de diversas abordagens na construção de um sistema mais complexo vem sendo estudada como Inteligência Artificial Híbrida. A ideia basicamente se baseia na união de diferentes técnicas, visando um sistema composto por partes com focos específicos balanceando assim seus benefícios e limitações individuais.

Como podemos ver em Bittencourt (1997), o desenvolvimento de uma arquitetura de IA híbrida pode ser baseada, por exemplo, na utilização de técnicas simbólicas para o raciocínio cognitivo responsável pela proatividade, objetivos e afins. Juntamente a isso, a lógica *Fuzzy* e Redes Neurais são encarregadas da reatividade do sistema, podendo ser implementado ainda um algoritmo de A*, por exemplo, para o processamento e construção de memória no chamado nível de instinto.

Com os avanços tecnológicos, as simulações computadorizadas foram ficando cada vez mais realistas, o que permitiu o desenvolvimento de conteúdos de entretenimento como os jogos *Microsoft Fly Simulator*, *Euro Truck Simulator* e *Farming Simulator*. Pesquisas e competições estão aproveitando isso permitindo que os participantes foquem apenas na implementação do software, o que abstrai as complexidades do desenvolvimento e de testes físicos em hardware. Na *Formula Student Driverless Simulation*, por exemplo, há o desenvolvimento de visão computacional para a captação da dados que serão utilizados em redes neurais e outras abordagens buscando a navegação autônoma (CONTRIBUTORS, 2020).

2.2 Agentes

Agentes são entidades inteligentes que são programadas com o intuito de realizar interações com o ambiente. Sua constituição se baseia basicamente de um mecanismo capaz de obter informações do ambiente que serão utilizadas na execução de um ciclo de raciocínio, gerando uma resposta que pode ser manifestada ou não por uma ação no ambiente, como observado na Figura 1. Podemos pensar, então, no agente como uma entidade que possui um certo nível de proatividade com o meio, pelo direcionamento de suas ações (FRANKLIN; GRAESSER, 1997).

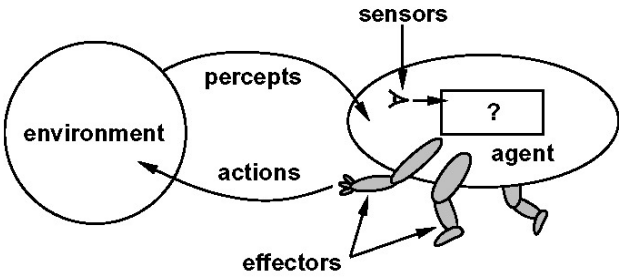

Figura 1 – Ilustração de um agente (RUSSELL; NORVIG, 2010)

O *Belief Desire Intention* (BDI) é uma abordagem que visa a modelagem de agentes baseada em conceitos do raciocínio humano. Como o próprio nome sugere, a metodologia é fundamentada em crenças, desejos e intenções, ou seja, o programador coloca uma gama de conhecimento base como as crenças que o agente possui do mundo, que são flexíveis e podem sofrer alterações. Além disso, há os desejos que são os objetivos do agente, ou seja, as metas que devem ser cumpridas nesse cenário. Baseando-se nisso, a execução de suas intenções é expressada por meio dos planos de execução, os quais são previamente programados (BRATMAN, 1987).

Nessa abordagem, podemos perceber que a principal diferença com relação a programas convencionais de computadores se encontra no fato de que não é algo totalmente fixo. Um agente inserido em um ambiente com um objetivo apenas executa os planos de acordo com suas crenças e desejos, que são influenciados de acordo com os valores obtidos. Ou seja, os algoritmos são mais flexíveis, podendo responder de forma diferente em decorrência de pequenas variações no estado. Devemos ressaltar também que nesse caso, uma sequência maior dos fatos segue sim uma ordem lógica, visto que os planos são previamente traçados. Entretanto, eles podem sofrer interrupções e interferências em decorrência de variabilidades externas.

O AgentSpeak é uma linguagem de programação na qual podemos programar um agente seguindo a metodologia BDI (RAO, 1996). JASON é um interpretador de AgentSpeak desenvolvido em Java que facilita a implementação de um agente por meio da definição de suas crenças, planos e objetivos (BORDINI; HÜBNER, 2007). Além do

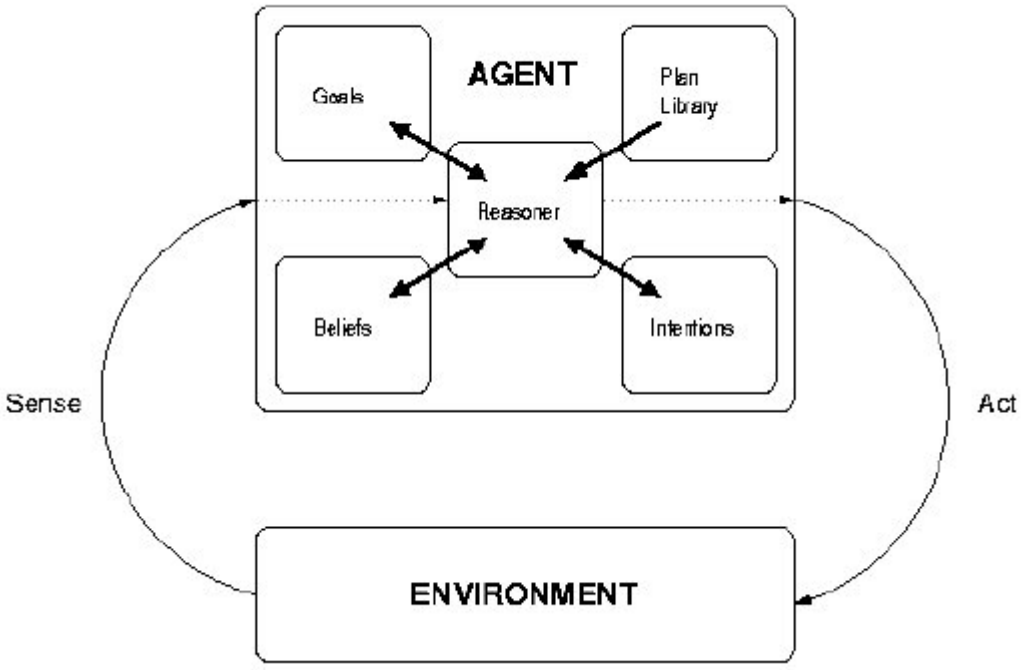

Figura 2 – Ilustração de arquitetura BDI (NORLING, 2001)

JASON, outras linguagens de programação são utilizadas na construção de agentes, cada uma com suas arquiteturas e modelagens, como é o caso do GOAL (BOER et al., 2002) e do ASTRA (DHAON; COLLIER, 2014).

2.3 Aprendizado de Máquina

Aprendizado de Máquina, ou *Machine Learning* (ML), é uma subdivisão do campo de IA não simbólica que visa o reconhecimento de padrões, a classificação ou a previsão de valores por meio dos dados obtidos utilizando conceitos estatísticos e matemáticos. Esse campo ganhou muita força nos últimos anos devido ao crescimento exponencial da internet em aparelhos capazes de captar diversas informações sem grandes limitações geográficas e temporais. A chegada do *Big Data* proporcionou desafios tecnológicos e influências significativas na sociedade, como é o caso da Lei Geral de Proteção de Dados Pessoais (LGPD). Esse fator, juntamente com o crescimento das capacidades computacionais e das melhorias de computação em nuvem, permitiu que algoritmos de ML desenvolvidos e conceituados durante anos de pesquisa na área pudessem ser implementados em questões reais com resultados satisfatórios (SARKER, 2021b).

Atualmente há estudos que visam melhorias nessa área, podendo trazer benefícios para diversos campos de conhecimento como saúde, jurídico, comércio e serviços. Várias questões foram surgindo, como a implementação de técnicas em Visão Computacional que superaram as limitações das abordagens clássicas, além do desenvolvimentos de

processamento de linguagem natural, do inglês, *Natural Language Processing* (NLP) (REYS et al., 2020).

2.3.1 Algoritmos de Aprendizado de Máquina

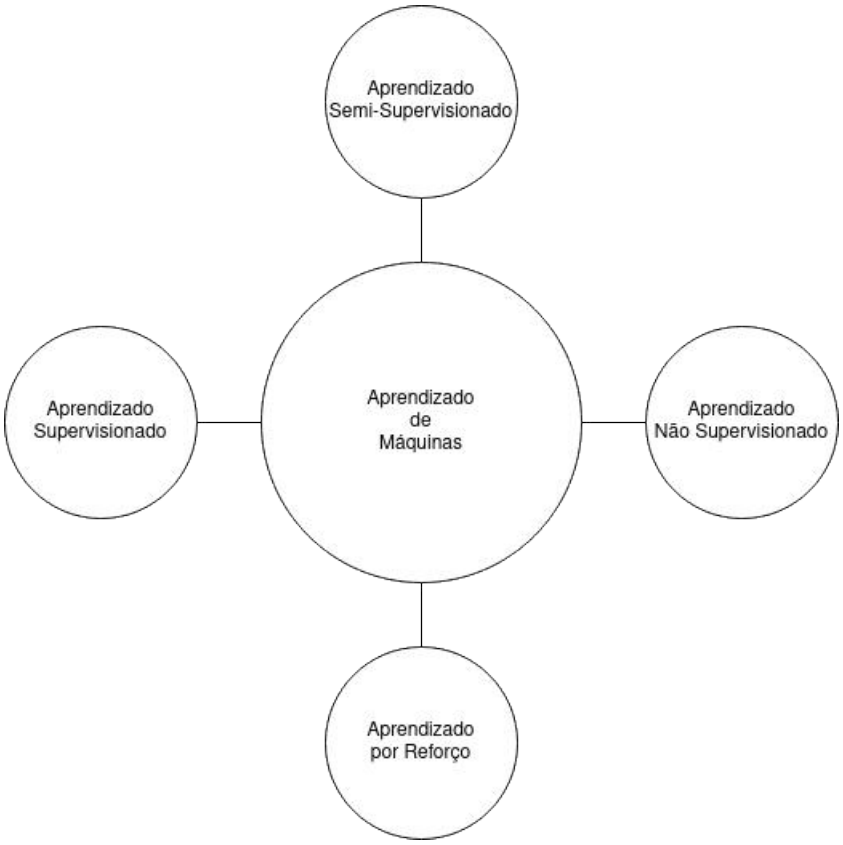

Figura 3 – Organograma de algumas áreas do Aprendizado de Máquinas

Como podemos observar em Sarker (2021b) e pela Figura 3, a área de ML possui várias abordagens e implementações, cada uma com suas vantagens em determinados tipos de problemas. Algumas técnicas utilizam algoritmos de aprendizado **supervisionados**, **não-supervisionados** e **semi-supervisionados**. No primeiro caso, há a definição do *objetivo* e das *características* encontradas no problema. Dessa forma, os algoritmos de regressão ou de classificação utilizam dessa informação para balancear os valores por meio de técnicas, como o método de gradiente descendente, fazendo uma conexão entre os dados de entrada e o resultado esperado a partir deles (HAN; PEI; KAMBER, 2011). No segundo caso, a diferença se encontra no fato de que não há a definição clara do "objetivo", deixando para que a IA encontre os padrões, resultando, por exemplo, em uma separação de grandes características que podem ser posteriormente rotuladas (HAN; PEI; KAMBER, 2011). Já no terceiro caso, há uma mistura da primeira com a segunda técnica, pois não há a necessidade de geração de todos os dados alvos necessários visto que a sua elaboração pode ser inviável por diversos fatores (HAN; PEI; KAMBER, 2011).

Concomitantemente, há técnicas que possuem conceitos e metodologias diferentes da anterior, como é o caso dos Algoritmos de Evolução Genética, baseados na evolução natural dos seres vivos. A ideia principal é constituída por cadeias de DNA sofrendo cruzamentos dos indivíduos que obtiveram a melhor performance em um ciclo de vida. Assim, os genes são perpetuados para as próximas gerações, podendo inclusive sofrer mutação decorrente de um fator X que visa dificultar a estabilidade e aumentar a diversidade (MITCHELL, 1998). Técnicas de Aprendizado por Reforço se baseiam em um conceito parecido de avaliação de performance no meio.

Já algumas abordagens buscam a implementação se baseando no cérebro. As redes neurais são constituídas de neurônios interconectados e ativados de acordo com uma função. Os pesos dessas funções podem ser alterados nos chamados treinamentos a fim de permitir o aprendizado da rede de acordo com o processamento de dados de entrada (*input*) e saída(*output*). A utilização de grandes e numerosas camadas de neurônios na tentativa de solucionar problemas mais complexos é chamada de Aprendizado Profundo, do inglês *Deep Learning* (SARKER, 2021a).

2.3.2 Aprendizado por Reforço

Aprendizado por Reforço é uma das técnicas de ML baseada em recompensas. Há um agente no ambiente recebendo informações, as quais são utilizadas e processadas na elaboração de uma resposta possível que é materializada em uma ação. Após isso, o agente percebe o efeito da ação no ambiente pela análise do retorno da função de recompensa, ponderando assim os pesos dos fatores de probabilidade de tal forma que permita um ajuste positivo com o avanço do treinamento (BARTO, 1998).

Diferentemente de algoritmos supervisionados, nos algoritmos por reforço, os agentes se adéquam ao meio com base em suas recompensas, o que se assemelha com o comportamento animal fisiológico. Dessa forma, em um determinado tempo t, o agente avalia sua situação no meio, bem como sua situação anterior a execução da ação a, efetuando a ponderação necessária na busca pela solução ótima ou semi-ótima.

Abordagens clássicas de Aprendizado por Reforço são baseadas em tabelas de probabilidade, como os algoritmos de QLearning e SARSA (SUTTON; BARTO, 2018). Ambos são bem parecidos, como podemos observar pelas equações representadas nas Figuras 4 e 5. De forma simplificada, uma tabela é criada possuindo respectivamente como linhas e colunas os estados e ações possíveis do ambiente. Os valores dessa tabela representam as probabilidades ponderadas, de acordo com as equações, de uma ação ocorrer. Quando um passo de treino é efetuado para um estado s_t, analisa-se a recompensa r_t resultante da ação a_t obtida por meio da *policy*, ponderando os valores do estado/ação de acordo com uma taxa de aprendizado α e uma taxa γ de importância das ações a futuras no estado s_{t+1}. Para o QLearning, pega-se sempre a melhor ação futura possível,

já para o SARSA, a ação é definida de acordo com uma *policy*.

$$Q^{new}(s_t, a_t) \leftarrow \overbrace{\underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left(\underbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}_{\text{new value (temporal difference target)}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)}^{\text{temporal difference}}$$

Figura 4 – Equação de atualização dos valores no algoritmo QLearning. [1]

$$Q^{new}(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_t + \gamma\, Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right]$$

Figura 5 – Equação de atualização dos valores no algoritmo SARSA [2]

Podemos perceber claramente aqui como um ambiente virtual de jogo se enquadra em um problema de Aprendizado por Reforço visto que possuímos cenários recebendo algum tipo de ação e manifestando seu impacto no sistema de pontos. Estendendo-se isso para questões de simulação de situações reais, conseguimos o treinamento de agentes, os quais podem ser posteriormente implementados fisicamente.

2.4 Visão Computacional

O ramo de visão computacional tem como objetivo a obtenção de informações em imagens. Inicialmente, no que chamamos hoje de abordagem clássica, os estudos se baseavam na manipulação das imagens por meio de técnicas de detecção de contornos, segmentação, filtros e afins, que geravam dados utilizados em modelos de classificação. No século 21, a utilização de Deep Learning começou a ganhar espaço após o vencimento da AlexNet na competição de visão computacional (KRIZHEVSKY; SUTSKEVER; HINTON, 2012). A ideia se baseava na *Convolutional Neural Networks* (CNNs), permitindo assim, a abstração dos problemas e aumentado o dinamismo das soluções, as quais eram antes extremamente específicas. Abordagens híbridas podem ser interessantes por aproveitarem a simplificação do processamento com técnicas clássicas além da generalização disponibilizada pelos algoritmos novos (MAHONY et al., 2019).

[1] Latex da equação obtido em: <https://en.wikipedia.org/wiki/Q-learning#Algorithm>
[2] Latex da equação obtido em: <https://en.wikipedia.org/wiki/State-action-reward-state-action#Algorithm>

3 Trabalhos Correlacionados

A junção de agentes com ML não é algo recente e vem sendo desenvolvida e testada ao longo do tempo. Visando analisar os trabalhos já existentes, foram pesquisadas em dispositivos de busca de trabalhos científicos, como o Google Scholar e ResearchGate, palavras chaves como: *Reinforcement Learning*; *BDI*; *Hibrid AI*; *Games*, encontrando assim, trabalhos interessantes que poderiam servir para a pesquisa. Os resumos foram então analisados a fim de entender as melhores correlações.

Podemos perceber por Krukoski (2019), a implementação de Aprendizado de Máquinas visando otimizar a decisão de agentes no momento de efetuar a ação do chute na *RoboCup*. Durante o artigo, é feito uma comparação entre os times com e sem a utilização das redes neurais, demonstrando uma maior vantagem no primeiro caso. Zatelli et al. (2012) possui um conceito parecido, sendo que houve a aplicação do algoritmo de QLearning na implementação do ataque do time.

Além dessa abordagem, percebemos também o estudo voltado para a construção estrutural do agente (BOSELLO, 2020) utilizando um framework chamado de **Jason-RL** (BOSELLO; RICCI, 2020). Nesse caso, a criação de agentes utiliza a linguagem Jason em conjunto com técnicas de *Reiforcement Learning* (RL) na elaboração de planos temporários, os quais podem determinar e ajustar planos visando o software 2.0, ou seja, o desenvolvimento de programa por meio de programas. A implementação, que foi o caso de teste do framework, focou na elaboração de um modelo de carro autônomo, sendo treinado primeiramente em um ambiente virtual passando posteriormente para um protótipo de carro construído. Os autores explicaram como a abordagem BDI ficou responsável pela elaboração do percurso como um todo, deixando o foco dos ajustes e da imprevisibilidade da pista para o RL. A proposta apresentada nesse trabalho se utiliza da proposta de Bosello e Ricci (2020) na elaboração de uma arquitetura que contêm o framework **Jason-RL** na construção de agentes voltados para jogos de *NES*, incluindo o gerenciamento do ambiente e da extração de informações.

Já em (NETO, 2016), há um trabalho de Inteligência Artificial em múltiplos níveis. A arquitetura BDI ficou responsável pelo nível cognitivo, o algoritmo de A* ficou responsável pelo nível instintivo e a lógica fuzzy ficou responsável pelo nível reativo. Dessa forma, o agente desenvolvido deveria participar de um jogo virtual simulando 'capture a bandeira', desenvolvendo estratégias e interpretando variáveis de entrada e saída.

No caso do *Multi-Agent Programming Contest* (MAPC), podemos considerar o ambiente simulado implementado como um jogo. Sendo assim, em Zatelli et al. (2013), percebemos uma abordagem puramente por IA simbólica com a utilização do JaCaMo.

Nesse caso, a modelagem e estruturação dos agentes, bem como suas comunicações e organizações, são definidas e planejadas, programando os agentes em JASON respeitando esse contexto de ambiente.

Com a evolução do poder computacional e do acesso a dados, abordagens puramente não simbólicas foram sendo desenvolvidas e se tornando cada vez mais comuns. Em Silver et al. (2016), há a utilização de técnicas de ML na elaboração de uma IA capaz de jogar o jogo GO. Por meio de aprendizado supervisionado, onde humanos vão ensinando e direcionando a IA, em conjunto a técnicas de RL, onde há a melhoria por meio de partidas autoadministradas, os conceitos foram sendo mapeados permitindo uma reposta aos movimentos do adversário.

Olhando agora para experimentos voltados para o jogo *Super Mario Bros*, observamos que Tsay, Chen e Hsu (2011), utilizaram técnicas de Aprendizado por Reforço no estudo e análise de abordagens para o desenvolvimento de um agente capaz de aprender por meio do ambiente do jogo. Já Lee et al. (2014) aplicou os processos de decisão de *Markov* visando entender as particularidades dos jogadores humanos por meio de técnicas de Aprendizado por Reforço Inverso, transmitindo tal conhecimento posteriormente para o agente.

Na visão do autor, a utilização pura de técnicas de ML no desenvolvimento de agentes em um ambiente virtual pode ser limitante, visto que muitas vezes as IA's são treinadas em cenários específicos. Há como contornar esses casos utilizando mais dados e poderes computacionais, porém isso pode aumentar consideravelmente o tempo de treinamento e processamento. Dito isso, uma abordagem utilizando o BDI, que possui um ciclo de raciocínio, no controle geral do agente em um ambiente de video game permite uma maior pró atividade, aumentando a chance de atuação em diversos cenários de um mesmo contexto. Aliado a isso, técnicas de Aprendizado por Reforço atreladas ao agente o tornam capaz de aprender como a mecânica de um jogo funciona, adaptando-se melhor ao meio otimizando suas ações.

A Tabela 1 apresenta um resumo com as características encontradas nos trabalhos correlacionados, posicionando essa proposta nesses atributos. Há a observação da utilização de Agentes e da modelagem BDI, bem como de ML e técnicas de RL. A possibilidade de desenvolvimento de IA Híbrida por outros meios nos leva à criação de outra coluna. As últimas 2 estão mais relacionados ao ambiente escolhido e a dinamicidade da implementação, ou seja, se o meio é um jogo e se o agente é capaz de elaborar novos planos de acordo com novos dados.

Trabalho	Agente	BDI	ML	RL	IA Híbrida	Jogo	Elaboração de planos
(KRUKOSKI, 2019)	X		X		X	X	
(Zatelli et al., 2012)	X	X	X		X	X	
(BOSELLO, 2020)	X	X	X	X	X		X
(BOSELLO; RICCI, 2020)	X	X	X	X	X		X
(NETO, 2016)	X	X			X	X	
(ZATELLI et al., 2013)	X	X				X	
(SILVER et al., 2016)			X	X		X	
(TSAY; CHEN; HSU, 2011)			X	X		X	
(LEE et al., 2014)			X	X		X	
Proposta do projeto	X	X	X	X	X	X	X

Tabela 1 – Características dos trabalhos correlacionados.

4 Desenvolvimento

É apresentado nesse capítulo, na seção 4.1, a abordagem inicial para entendimento geral da modelagem proposta, sendo especificada algumas ferramentas e decisões na seção 4.2. Por fim, a seção 4.3 apresenta o desenvolvimento da arquitetura, demonstrando as classes desenvolvidas e suas respectivas responsabilidades. É importante entender aqui os conceitos gerais, bem como as divisões dos módulos, pois dessa forma, aperfeiçoamentos e adaptações em determinadas áreas podem ocorrer sem grandes alterações nas demais.

4.1 Modelagem

O conceito do projeto se baseia na elaboração de uma arquitetura que implemente um agente capaz de entender e se adaptar a um ambiente de video game. Assim, a aprendizagem sobre o meio e a elaboração de planos de acordo com seus objetivos são importantes. O objetivo principal com o desenvolvimento desse projeto é o estudo e a implementação de um agente em um cenário que possa se aproximar a situações reais, visando que futuramente, esse trabalho possa servir de base na elaboração de robôs físicos capazes de interação com o mundo real. Com isso em mente, podemos pensar em uma modelagem abstrata sem se atentar a particularidade do fato de ser um ambiente virtual, permitindo assim a portabilidade do código principal do agente para outras situações caso seja interessante.

Na Figura 6, percebemos que a modelagem do projeto possui a extração de informações do estado atual do jogo, bem como de outros conceitos, que são utilizadas pelo agente no ciclo de raciocínio, acarretando em ações que são interpretadas e convertidas para comandos no meio.

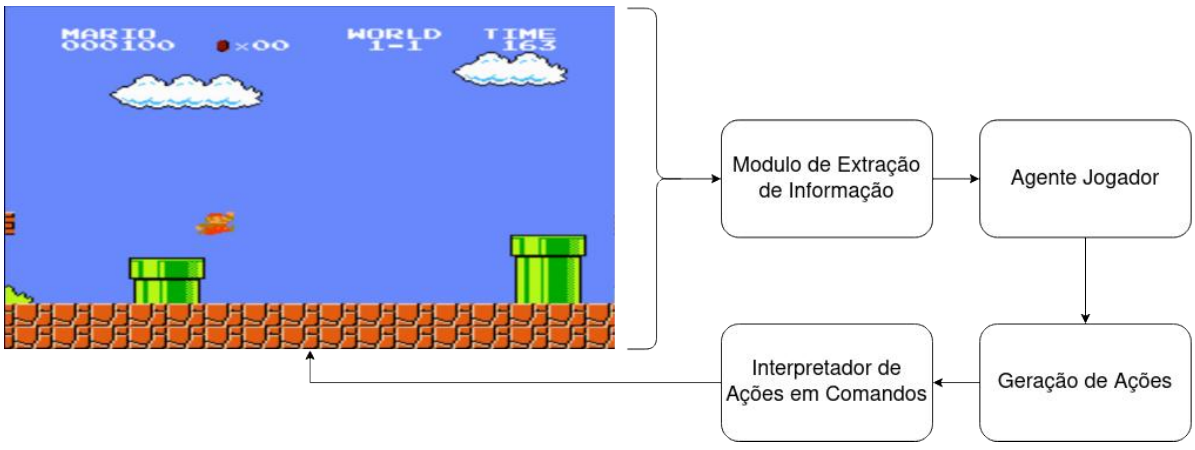

Figura 6 – Modelagem da implementação

4.2 Arquitetura

O entendimento do conceito geral da modelagem do problema permite redirecionar o foco às abordagens tecnológicas necessárias. Logo, como podemos observar na Figura 7, uma técnica de visão computacional foi utilizada na obtenção do estado atual do jogo. Em seguida, essas informações são convertidas em *JSON* para serem então interpretadas pelo agente, efetuando o ciclo de raciocínio que gera a ação ao meio, de acordo com a metodologia BDI apoiada pelo RL. O resultado então é redirecionado a um conversor que traduz as ações para comandos no jogo, permitindo assim a interação do agente com meio.

Como podemos observar na Figura 7 e como foi demonstrado na Figura 6, possuímos 3 grandes módulos para que tudo possa funcionar: um módulo dedicado ao ambiente do jogo, executando-o e obtendo informações específicas; um módulo dedicado a visão computacional responsável pela simplificação e extração do estado atual e por fim, um módulo de agente que entende, raciocina e gera a próxima ação do jogador.

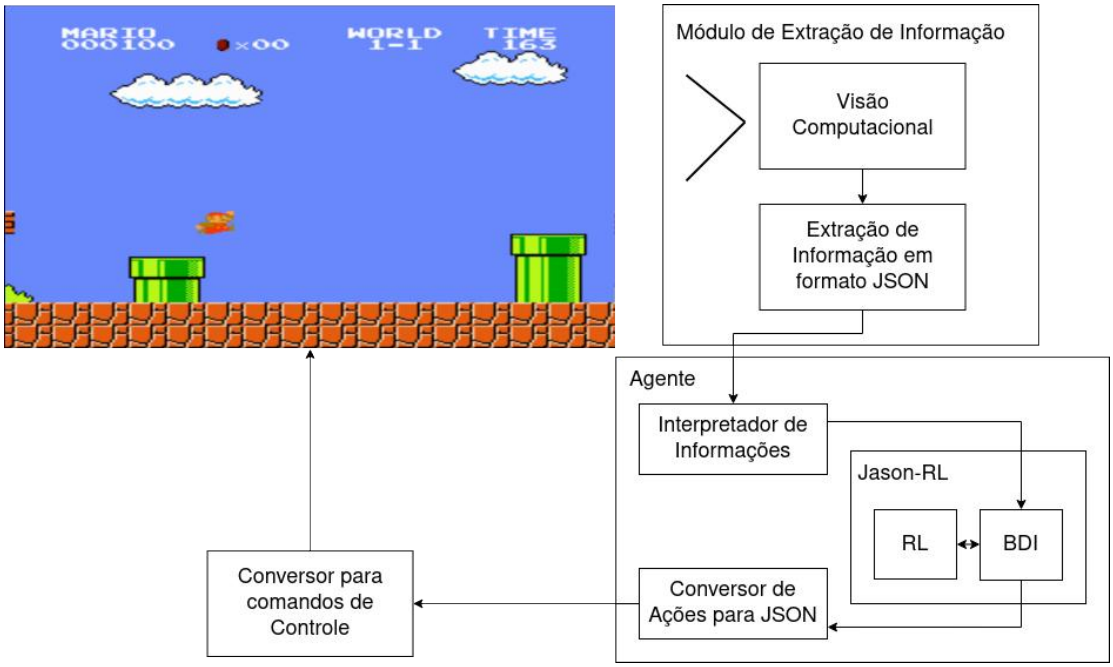

Figura 7 – Arquitetura da implementação

4.2.1 Linguagens, Modelo e Abordagem

No Capítulo 2, vimos a fundamentação teórica necessária para o desenvolvimento do trabalho. O foco nesse estudo foi dado às linguagens, modelos e técnicas que foram utilizados no decorrer da implementação. Optou-se pela utilização da modelagem BDI para agentes por meio da linguagem JASON, visto que o presente autor possui familiaridade pela participação no MAPC2020 (AHLBRECHT et al., 2020). A modelagem BDI facilita a elaboração dos planos para os contextos encontrados no jogo, definindo os conceitos

envolvidos e o fluxo do processamento. A linguagem JASON é uma implementação de AgentSpeak, que é composta por uma sintaxe e implementação de fácil entendimento. Além disso, JASON possui uma participação no JaCaMo, permitindo assim a expansão do projeto para um cenário multi-agent multi-dimensão caso seja interessante.

Outrossim, o conceito de Aprendizado por Reforço se enquadra bem em um ambiente virtual composto por recompensas e punições bem definidas, como é o caso de um jogo eletrônico. Logo, essa é uma boa abordagem de aprendizado de máquina a qual é integrada ao agente utilizando o framework **Jason-RL** (BOSELLO; RICCI, 2020). Alguns testes podem ser feitos em trabalhos futuros quanto às abordagens mais dinâmicas e responsivas de ML.

4.2.2 Framework

O framework utilizado durante o desenvolvimento do presente trabalho foi o **Jason-RL** desenvolvido por Bosello e Ricci (2020), uma ferramenta de uso geral permitindo a integração de Aprendizado por Reforço com a linguagem JASON. O programador define os conceitos gerais e alguns planos quando está desenvolvendo o agente, algumas informações então são injetadas e utilizadas pela implementação de RL, auxiliando assim o agente na elaboração de planos temporários que irão solucionar a tarefa atual. Dessa forma, são introduzidos o conceito de planos temporários e grau de confiança, utilizados nas avaliações que irão reforçar ou não as modificações efetuadas nos planos desenvolvidos. Esse trabalho aplica esse framework a um agente situado no jogo Super Mario Bros, de maneira que permita o aprendizado de como se deve jogar o jogo.

4.3 Kitsune AI

Kitsune AI foi o nome escolhido para se referir ao código desenvolvido ao longo do projeto, se referenciando às raposas japonesas que são conhecidas no folclore por sua inteligência e sabedoria.

A Programação Orientada a Objetos foi o paradigma escolhido para desenvolver o código. Ela permite uma melhor organização dos conceitos, necessário por conta da complexidade do projeto. Com isso, os módulos poderão ser facilmente aprimorados de forma independente, tornando o trabalho mais flexível. Dito isso, classes existem dentro da classe principal de controle (KitsuneAI), cada uma responsável pelo gerenciamento de um conceito demonstrado na seção 4.1: ambiente virtual (KitsuneEnv); visualização (KitsuneView) e agente (KitsuneAgent).

4.3.1 KitsuneEnv

Como comentado anteriormente, KitsuneEnv foi o nome escolhido para a classe responsável pelo gerenciamento do ambiente virtual e foi implementada utilizando o módulo Nes-py (KAUTEN, 2018a). Para tal, foi necessário o entendimento da renderização do jogo e do módulo *pyglet*[1] utilizado para controlar a interface gráfica. O módulo então foi construído possuindo a instância de outra classe responsável por definir algumas questões particulares para cada jogo escolhido. Nesse caso, para o jogo *Super Mario Bros*, o desenvolvimento se baseou na implementação de exemplo desenvolvido por Kauten (2018b) chamada de ***S**uper **M**ario **B**ros for **O**pen**AI** **G**ym*, que demonstra a manipulação da memória do jogo na obtenção de informações e na execução de ações como pular a tela inicial, sendo criado uma versão chamada de KitsuneSuperMarioBrosEnv. Na Figura 8, vemos a tela do jogo com objetos simplificados, uma das versões disponíveis (pixel).

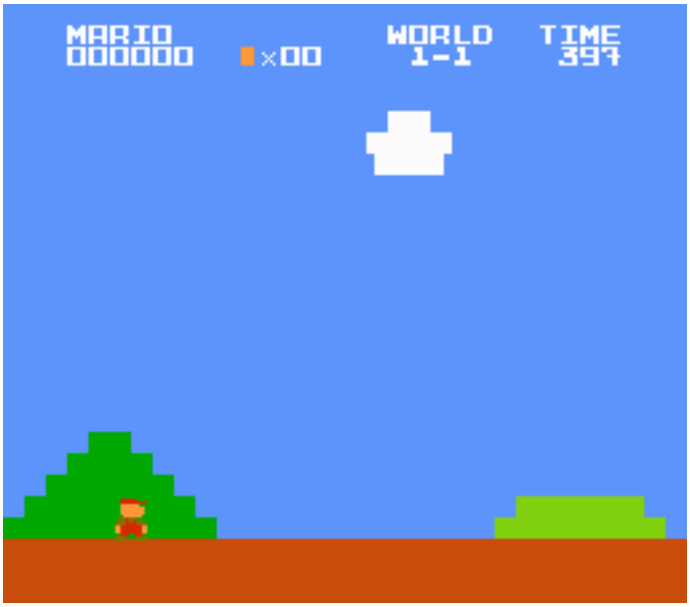

Figura 8 – Super Mario Bros, versão pixel (KAUTEN, 2018b)

Esse módulo gerencia a obtenção de algumas informações úteis do jogo, o controle de quadros por segundo, do inglês *Frames Per Second* (FPS), a leitura de informações do teclado e a interação com o jogo. Para isso, como podemos observar na Figura 9, uma classe específica dedicada a ROM(*Read-Only Memory*) de cada jogo, nesse caso a implementação KitsuneSuperMarioBrosEnv, deve ser desenvolvida ficando responsável pela obtenção de informações interessantes na construção do agente, como a quantidade atual de pontos, a movimentação do jogador, o estado final de uma fase e a imagem atual do jogo por exemplo. Outras métricas podem ser obtidas por meio de cálculos da posição de objetos efetuando comparações com os estados anteriores por exemplo, sendo necessário uma análise do que faz sentido para o jogo escolhido. Com isso, as informações podem

[1] Módulo python multiplataforma de desenvolvimento de jogos e aplicações visuais

ser utilizadas por um jogador humano por meio do teclado, no modo **manual**, ou pelo módulo KitsuneAgent apoiado pelo KitsuneView, no modo **automático**, para se definir uma ação a qual é traduzida pelo **nes-py** na execução da próxima etapa do jogo. A escolha do modo de jogo é definida quando se pressiona a tecla *M* e é definido o FPS por meio do escalonamento de uma função baseada no tempo.

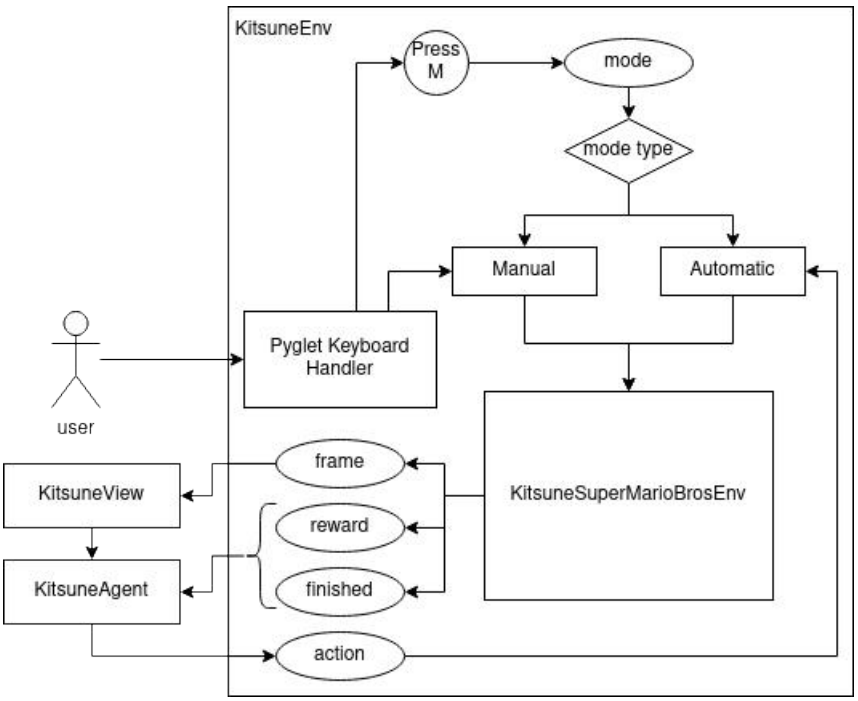

Figura 9 – Visão geral do módulo de ambiente

4.3.2 KitsuneView

KitsuneView é a classe responsável pela extração de informações relevantes da tela do jogo para serem utilizadas pela IA. Dessa forma, como podemos observar pela Figura 10, o principal objetivo aqui é detectar a posição do jogador na tela, bem como de elementos úteis como os inimigos, objetos especiais, blocos de interação, obstáculos, entre outros.

A abordagem adotada utiliza de métodos de Visão Computacional clássica, como apontado na seção 2.4. Baseando-se no trabalho de Morarji (2020), *Object Detection in Mario*, foram extraídos algumas imagens de elementos do jogo que foram utilizadas juntamente com o método de detecção de modelo do *opencv*[2]. Para isso, durante a inicialização do módulo, convertemos todas as imagens de objetos do jogo para tons de cinza, carregando-as na memória do computador para serem posteriormente utilizadas, armazenando também outras informações. Para cada objeto que desejamos encontrar, a função **opencv.matchTemplate()** do módulo *opencv* é utilizada para procurar alguma ocorrência na imagem atual do jogo, também convertida em tons de cinza, retornando o

[2] Biblioteca de tratamento e manipulação de imagens

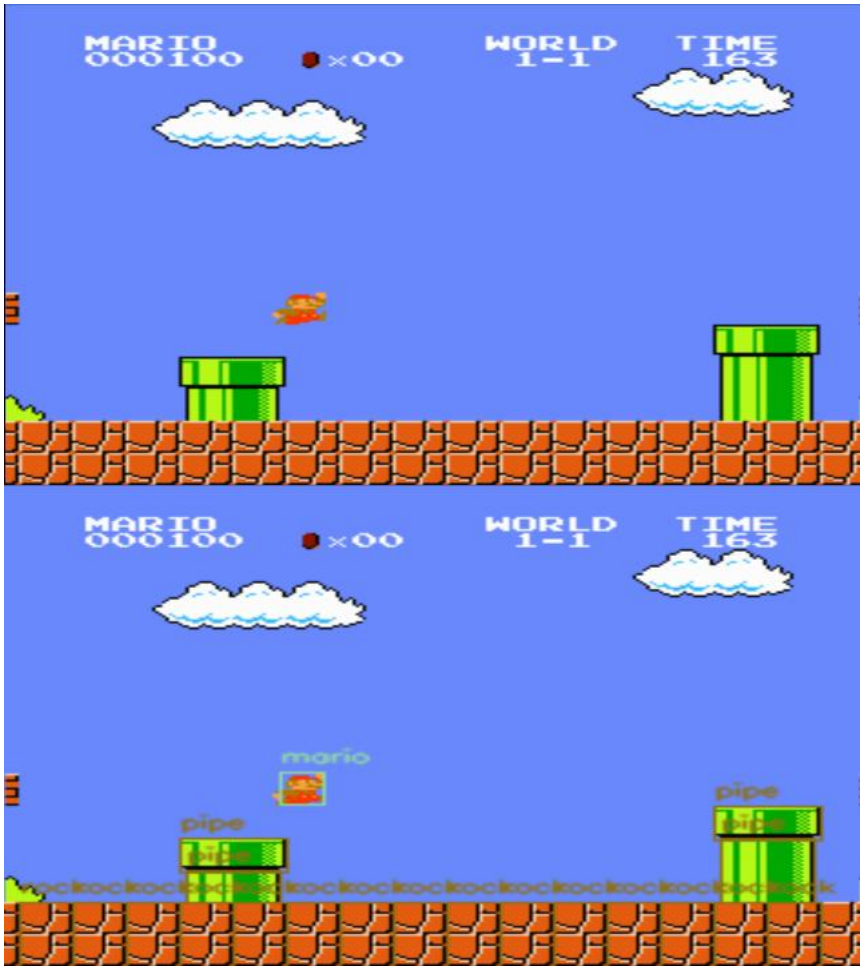

Figura 10 – Detecção de objetos no jogo

grau de semelhança encontrada nos pixels. O retorno dessa função é um conjunto de pontos indicando onde esses padrões foram encontrados. Podemos assim utilizar as informações armazenadas anteriormente como dimensão, nome, e tipo para gerar um dicionario com dados indicando o estado atual do jogo. Para acelerar o processo, a repetição dessa etapa para cada imagem de objeto que possuímos é realizada de forma paralela pelo módulo *multiprocessing*[3] do python. Além disso, de forma a simplificar algumas informações para o agente, pontos do mesmo objeto que estão localizados no mesmo local horizontalmente e que, verticalmente estejam contidos uns nos outros, são unidos a fim de se criar uma única informação. Na Figura 10 percebemos isso na detecção do *pipe*, que apresenta uma única peça central, independente da altura.

Essa abordagem pode não ser a melhor quando pensamos em complexidade computacional, podendo possuir outros métodos e técnicas que possuam resultados satisfatórios com complexidades menores. Para entender melhor isso, para cada imagem de objeto I, executamos uma busca utilizando a função **opencv.matchTemplate()**, além de um for para cada ponto J encontrado buscando concatenar as informações. O método

[3] Módulo python que disponibiliza ferramentas de programação paralela e concorrente

opencv.matchTemplate() utiliza do *Fast Fourier Transform*, que possui complexidade de $O(NlogN)$. Ou seja, de forma simplificada, a complexidade desse projeto é de $O(I \times J \times N \times logN)$, ou seja, $O(N^3logN)$.

4.3.3 KitsuneAgent

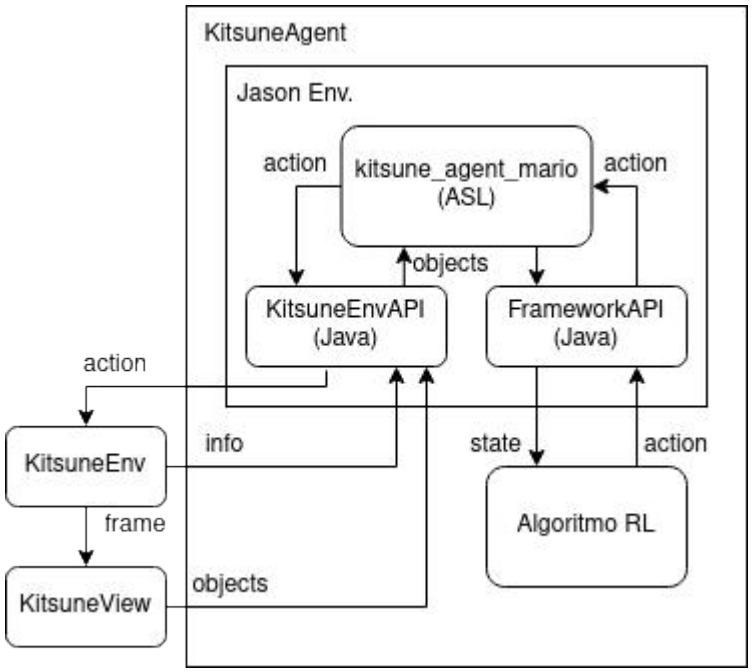

Figura 11 – Visão geral da arquitetura do agente.

Essa classe é responsável pela integração do agente BDI com os algoritmos de RL, bem como da utilização das classes citadas acima. Como podemos ver pela Figura 11, internamente essa classe possui uma referência tanto à KitsuneEnv quanto à KitsuneView instanciadas pela KitsuneAI. Quando um passo do processo ocorre, utilizando o módulo de ambiente, a KitsuneAgent pega as informações atuais do estado do jogo, resumindo-se na imagem atual, na recompensa e se é um estado final. Utilizando agora o módulo de visualização, são extraídos os objetos da cena, obtendo-se métricas adicionais pela classe do jogo que podem ser interessantes para o agente. No caso escolhido, as informações de velocidade dos objetos são importantes por demonstrarem, por exemplo, objetos relacionados ao cenário além de objetos que se movem, bem como o estado atual do Mario, se o mesmo está preso ou está se locomovendo, sendo obtidas realizando-se um cálculo simples comparando as posições dos objetos do estado atual com o estado anterior. Com essas informações, o processo segue por meio das comunicações da *Application Programming Interface*(API) com o cenário desenvolvido em Java que passará os objetos para o Cartago, podendo assim ser observado pelo Agente Jason, que utilizará novamente da API para obter as respostas da implementação de RL que resultará na próxima ação do jogo.

4.3.3.1 APIs

Jason foi desenvolvido utilizando a linguagem Java, permitindo assim que os ambientes desenvolvidos nessa linguagem possuam interações com os agentes implementados com AgentSpeak. Dessa forma, como a linguagem python foi escolhida para desenvolver a arquitetura, a comunicação por meio de APIs foi proposta por Bosello e Ricci (2020) e acatada nesse projeto. O módulo Flask permite a criação fácil de rotas que executam uma determinada função em uma porta, logo quando o módulo é criado, uma aplicação configura a porta 5003 para ser utilizada como canal de comunicação entre esses dois meios. Essa aplicação é executada em uma *Thread* dedicada para permitir que as demais partes do código sejam executadas. Aqui houve a necessidade de criação de Thread e não Processo, pois ainda há a necessidade de interação com os mesmos objetos instanciados inicialmente. Caso um Processo seja utilizado, ocorre uma cópia do estado da aplicação, dificultando o gerenciamento e controle de memoria compartilhada.

Diferentemente do framework utilizado, o qual instancia uma aplicação para os agentes implementados em python, porta 5002, e outra para o ambiente, porta 5003, optou-se pela utilização da mesma aplicação para ambas comunicações, diferenciando o tipo de função pelos caminhos onde:

- *'/**env**/<**string:env**>'* é utilizado para pegar informação do ambiente.
- *'/**env**/<**string:env**>/<**string:action**>'* é utilizado para efetuar uma ação no jogo.
- *'/**agent**/<**string:agent_id**>'* é utilizado para inicializar um algoritmo RL.
- *'/**agent**/<**string:agent_id**>/<**string:action_type**>'* é utilizado para pegar uma ação do algoritmo RL.

4.3.3.2 Java

Como comentado no tópico acima, precisamos implementar uma classe em Java que irá interagir com a API para gerenciar as observações no ambiente do agente Jason além de enviar os comandos para o jogo. Uma classe então foi desenvolvida baseada na *RestClient* disponibilizada pelo framework, que inicializa uma comunicação com a API em python e gerencia a conversão dos comandos do agente bem como atualiza as observações do mesmo.

Dessa forma, para executar alguma ação no jogo, as palavras indicando os botões apertados pelo controle do NES foram mapeados para seus respectivos equivalentes numéricos interpretados pelo KitsuneEnv, por exemplo o comando *"right"* é mapeado para o valor "1" e *"right+B"* para o valor "3"como podemos observar pela Tabela 2. Essas equivalências podem ser observadas no arquivo de configuração e referenciam na realidade

o index da lista de ações possíveis que o módulo Nes-py é inicializado. Os comandos de *"start"* e *"select"* não são levados em consideração pelo agente e o comando *"NOOP"* representa que nenhum botão foi acionado.

Ação	Index
NOOP	0
right	1
right + A	2
right + B	3
right + A + B	4
left	5
left + A	6
left + B	7
left + A + B	8
down	9
up	10
A	11
B	12
start	13
select	14

Tabela 2 – Mapeamento de ações no controle para equivalentes numéricos.

Além disso, foi desenvolvida uma função genérica dedicada a atualização de objetos observáveis pelo agente. Para todos os valores que o estado da API recebe do ambiente do jogo, é criada uma observação ***"object"*** com seus respectivos N parâmetros, além da atualização das observações de recompensa (***"reward"***) e pontuação (***"score"***). Antes de atualizar tais valores, ocorre uma limpeza removendo todas as ocorrências anteriores para que somente as informações mais recentes sejam consideradas pelo agente. Além disso, os únicos tratamentos especiais são com relação ao jogador, que possui uma observação dedicada chamada de ***"player"***, e com o ***"gameover"*** que só é lançado quando o estado atual é terminal. Pelo fato de que esses fatores são observáveis na maioria dos jogos, provavelmente essa parte não precisará ser muito alterada quando essa arquitetura for utilizada para outras implementações em outros cenários.

4.3.3.3 Jason

O desenvolvimento do agente Jason acaba sendo específico para o jogo escolhido. No caso desse trabalho, para o *Super Mario Bros*, algumas regras e ações foram desenvolvidas para que o agente consiga detectar informações relevantes do meio. Independente da utilização, uma questão importante que deve ser levada em consideração na hora de tomar decisões nesse jogo é descobrir no que o jogador está encostando. Logo, um conjunto de regras comparando as posições X e Y foram desenvolvidas, retornando a posição que os objeto 1 está tocando no objeto 2, no caso ***top, right, bottom, left***. Algumas outras

regras foram criadas utilizando essas como base para verificarem especificamente se o jogador está tocando algo e o local.

Utilizando essa informação, juntamente com outras condicionas, como por exemplo se é gameover, foram desenvolvidos ações que classificam determinados tipos de objetos em **"inimigos"** ou **"caminhos"**. Ou seja, adicionamos a crença de que um determinado tipo de objeto pode estar atrelado com certas características. Com isso, pensamos que ao passar o conjunto de crenças para os algoritmos de RL, uma relação de que objetos do tipo "inimigo"estão relacionados com a perda no jogo, e que por causa disso, qualquer objeto classificado com essa etiqueta deve ser evitado. A mesma ideia se aplica para os objetos os quais eu posso caminhar sobre, permitindo assim que eu prossiga pelo jogo.

Aqui percebemos o potencial de melhoria dessa abordagem, caso o algoritmo de aprendizado realmente perceba essa correlação, a classificação e/ou elaboração de planos abstratos criados pelo *Jason* podem antecipar determinados conhecimentos que auxiliaram no entendimento do jogo, podendo desempenhar melhor em fases distintas mas que possuem o mesmo contexto. Para que os algoritmos possam utilizar desses valores, foi necessário um mapeamento representado na Tabela 3 entre os nomes dos objetos para números a fim de serem utilizados nas matrizes de estado. O Código 4.1 conta com uma parte das regras e ações desenvolvidas para que o agente consiga detectar se um tipo de objeto pode ser classificado como inimigo ou não. Primeiramente definimos se um objeto 1 esta encostando em um objeto 2 caso haja sobreposição entre os valores de X e Y, respeitando os limites gerados pelas larguras W e alturas H as quais são utilizadas para definir o local de "encontro", possuindo valor *none* caso não estejam se encostando. A regra demonstrando se o jogador está encostando em um objeto é simplesmente uma simplificação da regra explicada anteriormente. Com isso, podemos definir se um objeto é do tipo inimigo caso a situação atual seja de *gameover* e o jogador esteja encostando em um objeto que não seja um caminho.

Nome	Número
block	0
brick	1
coin	2
flagpole	3
flower	4
g_mushroom	5
goomba	6
item	7
koopa	8
koopa_shell	9
l_mushrrom	10
mario	11
pipe	12
pipe_top	13
princess	14
question	15
rock	16
star	17
toad	18

Tabela 3 – Mapeamento de nome de objetos para números.

```
1  // Rule that get where an Object 1 is touching the Object 2
2  //the none value means that there is no touch in X or Y axis.
3  touching(X1, Y1, W1, H1, X2, Y2, W2, H2, DX, DY, PX, PY):-
4      x_range(X1, W1, X2, W2, DX, PX) &
5      (PX \== none) &
6      y_range(Y1, H1, Y2, H2, DY, PY) &
7      (PY \== none)
8  .
9  touching(X1, Y1, W1, H1, X2, Y2, W2, H2, 0, 0, none, none).
10
11 // Same as touching rule, but especific for player as object 1
12 player_touching(X, Y, W, H, DX, DY, TPX, TPY):-
13     player(_, PX, PY, PW, PH, PVX, PVY) &
14     touching(PX, PY, PW, PH, X, Y, W, H, DX, DY, TPX, TPY)
15 .
16 player_touching(X, Y, W, H, 0, 0, none, none).
17
18 // Using the position X and the width W of Object 1 and 2, determine
19 //where the second one is touching (PX) the first (left or right).
20 //Beside this, return the distance (DX) of this intersection.
21 x_range(X1, W1, X2, W2, ((X2+W2+1) - X1), right):-
22     (X1 >= X2) &
23     (X1 <= (X2+W2+1))
24 .
25 x_range(X1, W1, X2, W2, ((X1+W1+1) - X2), left):-
26     (X1 < X2) &
27     ((X1+W1+1) >= X2)
28 .
29 x_range(_,_,_,_,0, none).
```

```
30 // Using the position Y and the width W of Object 1 and 2, determine
31 //where the second one is touching (PY) the first (left or right).
32 //Beside this, return the distance (DY) of this intersection.
33 y_range(Y1, H1, Y2, H2, ((Y2+H2+1) - Y1), top):-
34     (Y1 >= Y2) &
35     (Y1 <= (Y2+H2+1))
36 .
37 y_range(Y1, H1, Y2, H2, ((Y1+H1+1) - Y2), bottom):-
38     (Y1 < Y2) &
39     ((Y1+H1+1) >= Y2)
40 .
41 y_range(_,_,_,_,0, none).
42
43 // Identifying Enemies:
44 // To be an Enemie, an object must being touching the player in
45 //X and Y position where the gameover observation is raised.
46 //Beside this we need to check if isn't an path type
47 +t_enemie(TN): true <- ?obj_name(TN, N); +enemie(N).
48 +enemie(N): true <- print("\nNew Enemie: ", N);.
49
50 // name, x, y, w, h, velocity_x, velocity_y
51 +object(TN, X, Y, W, H, VX, VY):
52     gameover &
53     player_touching(X, Y, W, H, DX, DY, TPX, TPY) &
54     (TPX \== none) & (TPY \== none) &
55     not t_path(TN)
56     <-
57     ?obj_name(TN, N);
58     print("\nENEMY: ",N);
59     +t_enemie(TN);
60 .
```

Código 4.1 – Parte do código do agente(kitsune_agent_mario.asl) que detecta inimigos

4.3.3.4 Algoritmo de Aprendizado por Reforço

O trabalho de Bosello e Ricci (2020) apresenta a possibilidade de implementação de algoritmos de RL tanto em java quanto em python. Como exemplo prático, o algoritmo SARSA foi implementado em Java por Bosello e Ricci (2020) e utilizado em uma das demonstrações do framework, sendo o primeiro algoritmo utilizado pelo agente no jogo *Super Mario Bros* para validação das outras partes da arquitetura. Além disso, os algoritmos QLearning e SARSA foram utilizados na verificação do desenvolvimento de algoritmos em python.

O framework é voltado para algoritmos da biblioteca TensorFlow, dessa forma, o conjunto de estados é criado pensando nessa utilização, sendo necessário o entendimento das transformações de estado para que os mesmos sejam transferidos para qualquer implementação de RL. Após efetuar a conexão por meio da API, verificou-se que o estado que chegava não condizia com toda a visualização encontrada pelo agente. Um estado estava reduzido a uma aparição de cada tipo de objeto observável, o que faz sentido nas

implementações de Bosello e Ricci (2020) pois somente há uma instância da observação no mundo. No caso de um jogo, pode haver nenhuma ou N instâncias do mesmo objeto observável, sendo necessário efetuar uma mudança no framework para obter todas as percepções do agente.

Foi acrescentado na primeira posição do array de cada caso do estado o tipo da percepção, que é simplesmente o número da ordem de aparição na configuração do algoritmo pelo agente. Com isso, uma lista de listas está sendo gerada e entregue pela API para o algoritmo de RL que gerencia esses valores de acordo com sua necessidade pois cada um possui especificações que podem ser facilitadas de acordo com a organização do estado.

Atrelado a isso, modificou-se a obtenção dos parâmetros de cada percepção para que valores nulos sejam aceitos, o que também pode ser uma realidade para os jogos e que não era contemplada pelo trabalho anterior. Por fim, foi necessário modificar a criação das ações possíveis no framework pois, como era baseado em um Set e não em uma Lista, a cada inicialização da aplicação, os valores numéricos referentes as ações variavam. Isso implica que o simples ato de salvar o estado do algoritmo de RL não garante os mesmos resultados quando se inicia novamente o jogo, visto que as ponderações não irão condizer com as ações treinadas anteriormente. Isso explica o por quê de quando se treinava o Mario para ir à direita, quando reiniciava a aplicação, ele ia para outro lado ou executava uma ação diferente. Atualmente as ações interpretadas pelo KitsuneEnv no jogo, as interpretadas pelo código Java, pelo agente e por fim pelo algoritmo de RL seguem os mesmos valores demonstrados na Tabela 2.

Para validação da arquitetura e das modificações necessárias, foram desenvolvidos, em python, os algoritmos de QLearning e SARSA explicados na seção 2.3.2, sendo implementado a *policy* ϵ-*greedy* a qual apresenta uma probabilidade ϵ de se ocorrer uma ação aleatória, escolhendo-se a melhor ação caso contrário. Atrelado a isso, o ϵ sofre um decaimento a cada passo, sendo seu valor multiplicado pelo ϵ-*decay*.

Na implementação, o estado primeiro é convertido de listas para tuplas, para permitir a utilização como chaves de dicionários, representando nossas tabelas, sendo posteriormente ordenados a fim de trazer o padrão de que tipos específicos de observações virão na frente de outras. Por fim conseguimos salvar esses valores por meio da biblioteca *pickle*[4] para garantir treinamento, utilização futura e observação de métricas.

4.3.3.5 Algoritmo de Evolução Genética

Embora o foco do trabalho seja Aprendizado por Reforço, de forma a verificar a flexibilidade do código desenvolvido, bem como outros possíveis resultados, foi implemen-

[4] Módulo python de serialização e deserialização binária.

tado um algoritmo simples de Evolução Genética baseada em Redes Neurais (MITCHELL, 1998). De forma simplificada e como demonstrado pela Figura 12, um organismo possui uma primeira camada de neurônios com $(256 * 240)/(16 * 16) * 9$ entradas levando-se em consideração que as imagens dos objetos possuem, normalmente, 16 pixels de largura e altura, e que a tela do jogo é de 256x240 pixels, cada um desses objetos possuem 9 informações ao todo, resultando no número total teórico de informação que pode ocorrer. Após isso, a rede segue com 3 camadas de 16 neurônios e finaliza com uma camada de 12 neurônios demonstrando as opções de ações finais sendo ativados com *softmax* visando apenas uma ativação. Todas as camadas são totalmente conectadas. Uma população de 50 organismos foi escolhido para representar uma geração, possuindo uma taxa de 0.5 de mutação com decaimento de 0.9 a cada geração. Para gerenciar o estado recebido do JASON, que é em formato matricial, efetuamos simplesmente uma ordenação seguido de redução para 1 dimensão.

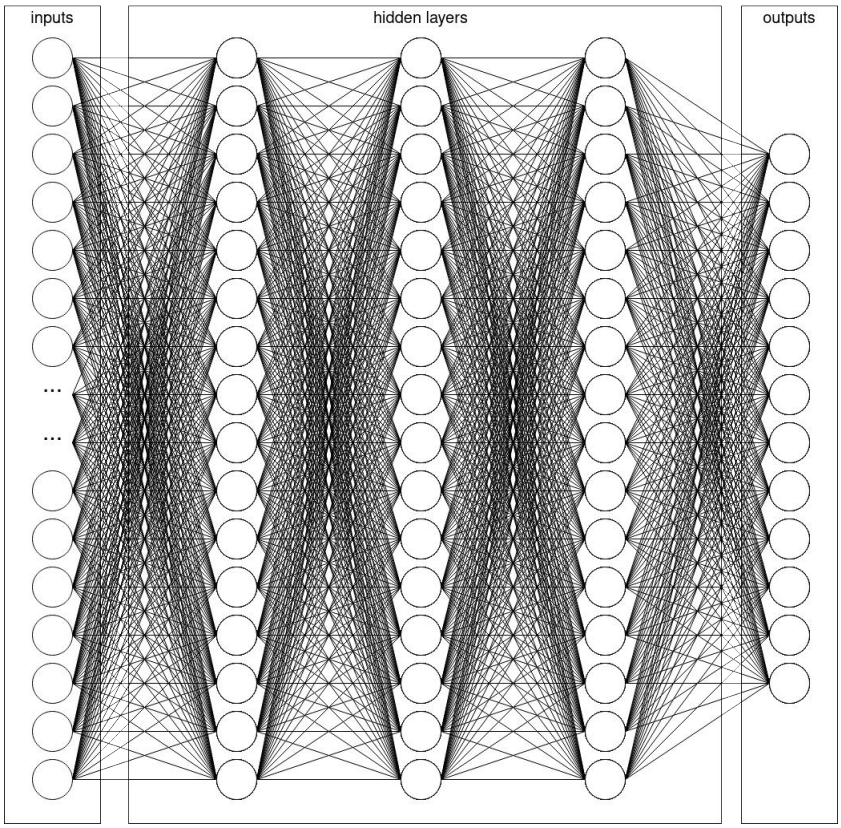

Figura 12 – Demonstração da Rede Neural desenvolvida para o Organismo.

Quando o código começa a ser executado, uma nova população é desenvolvida e o primeiro organismo é eleito para decidir as ações no ambiente. Se a cada 100 decisões não houver uma melhoria na recompensa acumulada, então força-se um reset do sistema para ser avaliado o próximo organismo. Quando todos os membros da geração são avaliados, pega-se os 10% que melhor pontuaram e cruza-os com os demais indivíduos seguindo uma probabilidade de escolha baseada no desempenho, criando-se assim a próxima geração.

Atrelado a isso, cada "DNA", no caso representado pelas redes neurais, possui uma probabilidade de sofrer mutação seguindo um desvio do valor herdado pelos pais. Por fim o módulo *dill*[5] é utilizado para salvar os treinos, o que permite a comparação, análise e reprodutibilidade futura.

Os códigos de aprendizado podem ser facilmente escolhidos alterando-se o parâmetro no agente JASON, influenciando na instância do algoritmo demonstrado pelo Código 4.2. Perceba que aqui podemos definir tanto algoritmos de Aprendizado por Reforço como outras implementações, como por exemplo Evolução Genética, demonstrando mais uma vez a modularidade do código desenvolvido.

```
1  def route_agent_info(self, agent_id:str):
2      json_data = request.get_json(force=True)
3      actions = list(range(json_data['a_min'][0], json_data['a_max'][0]+1))
4      parameters = json_data['parameters']
5
6      print("Agent".center(30,"-"))
7      print(f"type: {json_data['agent_type']}")
8      print(f"actions: [{min(actions)}, {max(actions)}]")
9      print(f"parameters: \n{json.dumps(parameters, indent=4)}")
10
11     if json_data['agent_type'] == "qlearning":
12         self.agent = QLearning(
13             len(actions)-1, #Removing reset
14             **parameters
15         )
16     if json_data['agent_type'] == "py_sarsa":
17         self.agent = Sarsa(
18             len(actions)-1, #Removing reset
19             **parameters
20         )
21     if json_data['agent_type'] == "natural_evolution":
22         self.agent = NaturalEvolution(
23             len(actions)-1, #Removing reset
24             **parameters
25         )
26     self.info["algorithm"] = json_data["agent_type"]
27
28     print("Successfully Loaded".center(30,"-"))
29
30     return {}
```

Código 4.2 – Parte do código da (KitsuneAgent) que define o algoritmo a ser utilizado

[5] Módulo python de serialização e deserialização binária.

5 Resultados e Discussões

O propósito desse trabalho foi o estudo de diversas frentes da área de Inteligência Artificial para implementar uma arquitetura híbrida capaz de jogar um jogo eletrônico. Como podemos observar nas seções 4.2 e 4.3, a quebra do problema em três grandes módulos permite que alterações específicas ocorram sem que mudanças significativas nas demais frentes sejam necessárias.

Além disso, como comentado anteriormente, foi desenvolvido um agente capaz de jogar *Super Mario Bros* que identifica os conceitos de "inimigo" e "caminho". Com isso, conseguimos atrelar a um agente, que aprendeu a se locomover no meio utilizando técnicas de aprendizado por reforço e evolução genética, conceitos simbólicos e de "alto" nível os quais nós humanos normalmente desenvolvemos ao jogar esses tipos de jogos. A arquitetura implementada permite uma fácil troca entre os algoritmos de RL, permitindo a comparação entre diferentes técnicas como foi ilustrado pelo QLearning e SARSA, sendo possível inclusive o estudo de outras abordagens de Aprendizado de Máquinas, como demonstrado pela implementação do algoritmo de Evolução Genética com Redes Neurais. Não obstante, há a possibilidade de experimentar outros meios de detecção de objetos que podem ser avaliados. Ademais, outros jogos podem ser implementados e validados, necessitando de poucas alterações no código. Isso demonstra como foi possível o desenvolvimento de uma arquitetura modular que pode servir de base para trabalhos futuros.

Algoritmos clássicos de RL não apresentam grandes resultados em cenários complexos como é o caso do *Super Mario Bros*. Como podemos observar nas Figuras 13 e 14, com mais de 430 episódios de treinamento, percebemos que não houve um aumento significativo na recompensa acumulada. Afora, após aproximadamente 250 episódios, algo de estranho começa a ocorrer nos algoritmos deixando os valores extremamente baixos em certos momentos. Tirando isso, há uma convergência para valores próximos a 100, não alcançando por exemplo valores significativamente maiores. O melhor cenário do algoritmo SARSA, por exemplo, com pontuação acumulada de 139 não passou do primeiro inimigo. Esse resultado pode ter sido por conta dos parâmetros escolhidos: alpha = 0.75; gamma = 0.8; epsilon = 0.9; epsilon_decay = 0.95; epsilon_min = 0, os quais foram escolhidos de forma arbitrária seguindo simplesmente o significado dos parâmetros, sem efetuar uma otimização aprofundada. Mesmo assim, isso não resolve o problema da falta de aproveitamento de conhecimento de estados parecidos, o que as implementações utilizadas não levam em consideração. Isso pode ter dificultado a convergência do algoritmo em decorrência da grande quantidade de estados possíveis juntamente com o número significativo de ações possíveis.

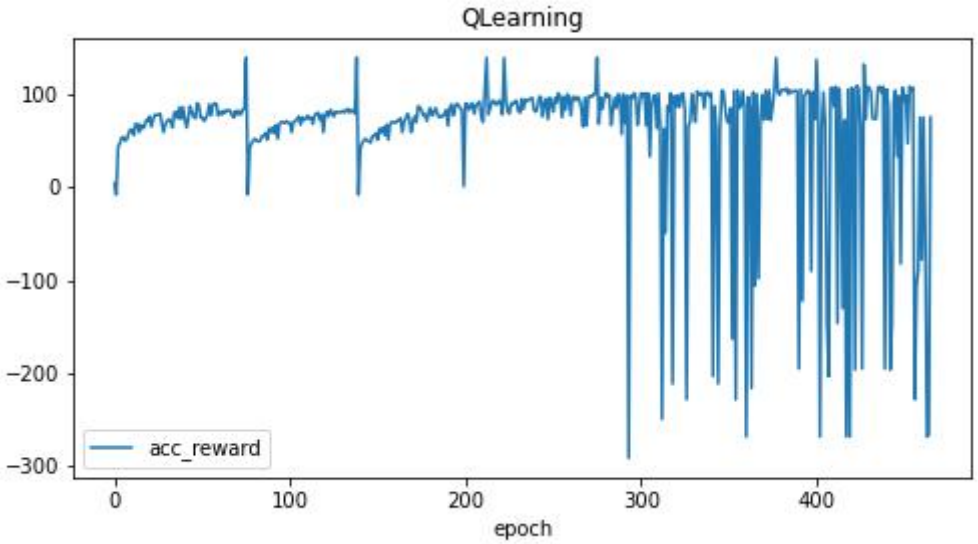

Figura 13 – KitsuneAI com QLearning executado por 465 episódios.[1]

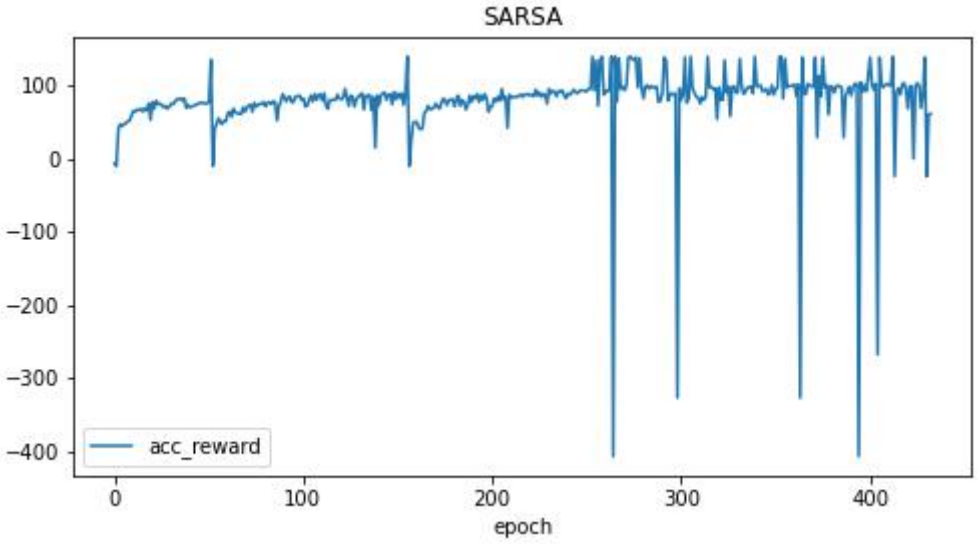

Figura 14 – KitsuneAI com SARSA executado por 432 episódios.[1]

Algoritmos que possuem funções sendo ponderadas de acordo com os valores encontrados conseguem criar relações entre os números apresentados de forma mais interessante do que as abordagens clássicas de RL. No caso do algoritmo de Evolução Genética de uma Rede Neural, podemos perceber pela Figura 15 que o melhor organismo da primeira geração já possuía resultados acima de 100, possuindo um pico acima de 600. Ao analisar os gráfico percebemos uma tendência crescente dos valores, diferentemente das outras abordagens apresentadas as quais se estabilizaram. Vale ressaltar aqui que cada geração possui 50 organismos analisados, e que alterando algumas questões como taxa de mutação, decaimento da mutação, número da população, método de cruzamento e até

[1] **acc_reward** no eixo y representa a recompensa acumulada por época apresentada pelo eixo x **epoch**.

mesmo a topologia da rede, resultados melhores poderiam ser obtidos visto que os valores também foram escolhidos de forma arbitrária. Diferentemente das abordagens anteriores, cada ação definida pelo agente é executada por 3 passos consecutivas, simplificando a quantidade de informação recebida.

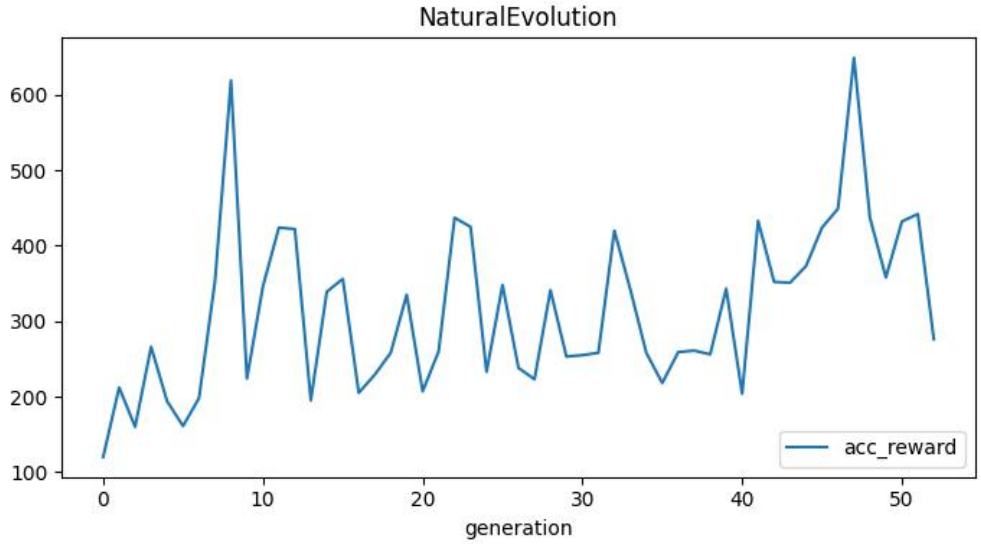

Figura 15 – KitsuneAI com Evolução Genética de Rede Neural executado por 52 gerações.[2]

Durante o desenvolvimento do KitsuneView, a dificuldade de encontrar, gerenciar e verificar as versões corretas do tensorflow, CUDA, python e outras bibliotecas, trouxe uma maior complexidade na verificação das técnicas. Algumas versões pararam de oferecer suporte, como é o caso do tensorflow 1.13 para o python 3.8.

A primeira técnica que foi utilizada para testar a detecção de objetos na tela foi a utilização de algoritmos Auto-Supervisionados, do inglês *Self-Supervised*. A ideia seria implementar o *DINO*, o qual iria segmentar automaticamente os objetos da imagem desenvolvendo mapas de calor para os objetos relevantes (CARON et al., 2021). A utilização de tal metodologia necessitava de aproximadamente 25GB de RAM para rodar os vídeos de teste, o que inviabilizou a aplicação devido as limitações computacionais.

Como segunda tentativa, algoritmos não supervisionados foram analisados para possíveis implementações. A vantagem dessa abordagem é a não necessidade de criação de rótulos para que seja possível detectar os objetos na tela, possibilitando uma generalização mais fácil para aplicações em outro cenário. *SPAIR* foi a primeira verificação para tentativa dessa abordagem em nosso problema. Analisando o paper de Crawford e Pineau (2019), podemos perceber que a proposta se sai relativamente bem, comparado com os outros algoritmos mencionados como o *AIR*. Além disso, percebemos uma boa detecção em jogos de Atari, o que traz a implementação para perto do nosso universo. O desenvolvimento dessa

[2] **acc_reward** no eixo y representa a recompensa acumulada pelo melhor organismo da geração apresentada pelo eixo x **generation**.

ferramenta foi efetuada em python 3.8 com a versão de tensorflow 1.13.2, a comunidade de python não da mais suporte a essa versão de tensorflow para o python 3.8, o que gerou uma grande dificuldade na instalação dos requisitos para a execução do teste. Após a execução de algumas épocas, observou-se que o algoritmo não iria possuir um bom desempenho para algo mais complexo como é o caso do *Super Mario Bros*, que possui mais elementos de plano de fundo em contraste com os jogos de Atari. *SILOT* (CRAWFORD; PINEAU, 2020) foi testado logo após, visto que foi uma implementação do *SPAIR* em vídeos que apresentavam objetos se movimentando, chegando-se na mesma conclusão.

SPACE (LIN et al., 2020) foi analisado como uma possível implementação para nosso problema. Utilizando uma abordagem interessante de separação de contexto, entre o que seria considerado o plano de fundo e o plano principal da imagem, essa técnica de segmentação apresentava resultados interessantes também em imagens de jogos eletrônicos do Atari. Os resultados, como apontado pelos próprios autores, não seriam satisfatórios para a exportação de informações relevantes para algoritmos de RL.

Por fim, *GMAIR* (ZHU et al., 2021) foi o último algoritmo analisado para ser utilizado em nossa implementação. Como uma evolução do *SPAIR* e *SILOT*, trazendo a ideia de utilização de um Operador de Mistura Gaussiana(*Gaussian Mixture*), essa implementação conseguiria, além de detectar e representar os objetos de uma tela, classificar os seus tipos, já efetuando a separação das detecções em classes simbólicas para o algoritmo. Isso trás uma possibilidade interessante para o projeto por permitir que uma classificação por semelhança seja feita, fazendo com que a IA crie seus próprios rótulos para os objetos "enxergados" na tela. Tal abordagem não conseguiu, a princípio, lidar com um cenário mais complexo como o do jogo proposto.

6 Conclusão

O presente trabalho apresentou informações relacionadas a diferentes frentes da área de Inteligência Artificial, servindo como referência para o entendimento dessas áreas e suas aplicações, bem como do desenvolvimento de uma arquitetura híbrida e modularizada que permite a criação de um agente capaz de aprender a jogar jogos de NES. Como podemos observar no capítulo 5, algumas abordagens na área da visão computacional foram experimentadas, demonstrando que podemos alterar essa parte da implementação sem impactar significativamente os demais módulos. Além disso, podemos expandir os algoritmos de RL que podem ser instanciados pela KitsuneAgent. Dito isso, foram pesquisados as áreas de Aprendizado por Máquinas bem como algoritmos de Aprendizado por Reforço e Agentes BDI, entendendo arquiteturas e modelagens existentes nesses casos. Após levantamento do estado da arte em AI híbrida, analisando projetos que apresentam diferentes objetivos e metodologias, elaborou-se um modelo e uma arquitetura que representa as grandes divisões e responsabilidades necessárias. Com isso, foi desenvolvido um agente BDI juntamento com algoritmo de RL ou Evolução Genética que permite o aprendizado do jogo do Mário, o qual consegue detectar obstáculos, inimigos, caminhos e outros objetos na tela. A implementação e a modularização da proposta se demonstrou escalável e de fácil alteração como demonstrado pela implementação de outra técnica de aprendizado. O código fonte pode ser encontrado no repositório do github Kitsune_AI[1].

Analisando as abordagens de Aprendizado por Reforço clássico e Evolução Genética com Redes Neurais, percebemos um melhor desempenho na última. Isso indica que algoritmos mais modernos tendem a entender melhor as correlações dos valores extraídos pela técnica BDI e de visão computacional, podendo inclusive tornar o aprendizado relativamente genérico dentro do contexto do jogo, o qual pode ser validado efetuando-se um treinamento em uma fase e o teste em outra.

6.1 Trabalhos Futuros

Futuramente, algumas técnicas de Visão Computacional podem ser implementadas e avaliadas para a detecção de objetos do jogo. Algoritmos supervisionados podem trazer bons resultados e com um desempenho melhor do que o obtido, visto que há arquiteturas voltadas à detecções em tempo real, como por exemplo a YOLO (REDMON et al., 2015).

Além disso, não foram explorados muitos algoritmos de aprendizado por reforço,

[1] Github: <https://github.com/robsonzagrejr/kitsune_ai>

podendo ser aplicadas técnicas mais sofisticadas que agregam redes neurais, utilizando-se da arquitetura desenvolvida para identificar a melhor técnica para o cenário do jogo. Não obstante, melhores técnicas de Evolução Genética podem ser testadas, como por exemplo evoluções de topologia de rede (STANLEY; MIIKKULAINEN, 2002). Uma ampliação da arquitetura para múltiplas execuções simultâneas pode ser implementada, o que se resumiria em várias instâncias da classe KitsuneAI.

Por fim, a arquitetura pode ser utilizada em outro jogo de NES a fim de verificar sua flexibilidade e modularização. O intuito da arquitetura é facilitar a modificação de cada módulo para reduzir as configurações necessárias nos demais, respeitando, obviamente, as implementações específicas existentes, como é o caso dos planos do agente Jason.

Referências

AHLBRECHT, T. et al. The multi-agent programming contest: A résumé. In: AHLBRECHT, T. et al. (Ed.). **The Multi-Agent Programming Contest 2019**. Cham: Springer International Publishing, 2020. p. 3–27. ISBN 978-3-030-59299-8. Citado na página 34.

BARTO, R. S. S. A. G. Adaptive computation and machine learning. In: **Adaptive computation and machine learning.** [S.l.]: Cambridge, Mass. : MIT Press, ©1998., 1998. ISBN 0262193981, 9780262193986. Citado 2 vezes nas páginas 19 e 27.

BITTENCOURT, G. In the quest of the missing link. In: **IJCAI**. [S.l.: s.n.], 1997. Citado na página 23.

BOER, F. S. de et al. Agent programming with declarative goals. **CoRR**, cs.AI/0207008, 2002. Disponível em: <https://arxiv.org/abs/cs/0207008>. Citado na página 25.

BORDINI, R. H.; HÜBNER, J. F. Jason a java-based interpreter for an extended version of agentspeak developed by. In: . [S.l.: s.n.], 2007. Citado na página 24.

BOSELLO, M. **Integrating BDI and Reinforcement Learning: the Case Study of Autonomous Driving**. Dissertação (Mestrado) — University of Bologna, 2020. Disponível em: <http://amslaurea.unibo.it/21467/>. Citado 2 vezes nas páginas 29 e 31.

BOSELLO, M.; RICCI, A. From programming agents to educating agents – a jason-based framework for integrating learning in the development of cognitive agents. In: DENNIS, L. A.; BORDINI, R. H.; LESPÉRANCE, Y. (Ed.). **Engineering Multi-Agent Systems**. Cham: Springer International Publishing, 2020. p. 175–194. Citado 7 vezes nas páginas 19, 29, 31, 35, 40, 44 e 45.

BRATMAN, M. **Intention, Plans, and Practical Reason**. [S.l.]: Cambridge: Cambridge, MA: Harvard University Press, 1987. Citado 2 vezes nas páginas 19 e 24.

CARON, M. et al. Emerging properties in self-supervised vision transformers. **CoRR**, abs/2104.14294, 2021. Disponível em: <https://arxiv.org/abs/2104.14294>. Citado na página 51.

CONTRIBUTORS, F. **Formula Student Driverless Simulation**. [S.l.]: GitHub, 2020. <https://github.com/FS-Driverless/Formula-Student-Driverless-Simulator>. Citado na página 23.

CRAWFORD, E.; PINEAU, J. Spatiial invariant unsupervised object detection with convolutional neural networks. In: **Thirty-Third AAAI Conference on Artificial Intelligence**. [S.l.: s.n.], 2019. Citado na página 51.

CRAWFORD, E.; PINEAU, J. Exploiting spatial invariance for scalable unsupervised object tracking. In: **Thirty-Fourth AAAI Conference on Artificial Intelligence**. [S.l.: s.n.], 2020. Citado na página 52.

DHAON, A.; COLLIER, R. W. Multiple inheritance in agentspeak(l)-style programming languages. In: **Proceedings of the 4th International Workshop on Programming Based on Actors Agents amp; Decentralized Control**. New York, NY, USA: Association for Computing Machinery, 2014. (AGERE! '14), p. 109–120. ISBN 9781450321891. Disponível em: <https://doi.org/10.1145/2687357.2687362>. Citado na página 25.

FRANKLIN, S.; GRAESSER, A. Is it an agent, or just a program?: A taxonomy for autonomous agents. In: MÜLLER, J. P.; WOOLDRIDGE, M. J.; JENNINGS, N. R. (Ed.). **Intelligent Agents III Agent Theories, Architectures, and Languages**. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997. p. 21–35. ISBN 978-3-540-68057-4. Citado na página 24.

HAN, J.; PEI, J.; KAMBER, M. **Data mining: concepts and techniques**. [S.l.]: Elsevier, 2011. Citado na página 26.

KAUTEN, C. **Nes-py**. 2018. GitHub. Disponível em: <https://github.com/Kautenja/nes-py>. Citado na página 36.

KAUTEN, C. **Super Mario Bros for OpenAI Gym**. 2018. GitHub. Disponível em: <https://github.com/Kautenja/gym-super-mario-bros>. Citado 2 vezes nas páginas 9 e 36.

KRIZHEVSKY, A.; SUTSKEVER, I.; HINTON, G. E. Imagenet classification with deep convolutional neural networks. In: PEREIRA, F. et al. (Ed.). **Advances in Neural Information Processing Systems**. Curran Associates, Inc., 2012. v. 25. Disponível em: <https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf>. Citado 2 vezes nas páginas 19 e 28.

KRUKOSKI, F. M. ImplementaÇÃo e anÁlise de diferentesalgoritmos de aprendizado de mÁquina noambiente 2d simulado da robocup. 2019. Citado 3 vezes nas páginas 19, 29 e 31.

LEE, G. et al. Learning a super mario controller from examples of human play. In: **2014 IEEE Congress on Evolutionary Computation (CEC)**. [S.l.: s.n.], 2014. p. 1–8. Citado 2 vezes nas páginas 30 e 31.

LIN, Z. et al. Space: Unsupervised object-oriented scene representation via spatial attention and decomposition. In: **International Conference on Learning Representations**. [s.n.], 2020. Disponível em: <https://openreview.net/forum?id=rkl03ySYDH>. Citado na página 52.

MAHONY, N. O. et al. Deep learning vs. traditional computer vision. **CoRR**, abs/1910.13796, 2019. Disponível em: <http://arxiv.org/abs/1910.13796>. Citado na página 28.

MITCHELL, M. **An Introduction to Genetic Algorithms**. Cambridge, MA, USA: MIT Press, 1998. ISBN 0262631857. Citado 2 vezes nas páginas 27 e 46.

MORARJI, V. **Object Detection in Mario**. 2020. GitHub. Disponível em: <https://github.com/vmorarji/Object-Detection-in-Mario>. Citado na página 37.

NETO, J. R. V. **COMPOSIÇÃO DE TÉCNICAS DE INTELIGÊNCIA ARTIFICIAL EM UMA ARQUITETURA MULTI NíVEL PARA EMERGÊNCIA DE COMPORTAMENTOS EM AGENTES COOPERATIVOS**. 2016. Citado 2 vezes nas páginas 29 e 31.

NILSSON, N. **The quest for artificial intelligence: A history of ideas and achievements**. [S.l.: s.n.], 2010. ISBN 9780521122931. Citado na página 23.

NORLING, E. Flexible, reusable agents for modelling human operators (extended abstract). 04 2001. Citado 2 vezes nas páginas 9 e 25.

PAPADIMITRIOU, C. Turing ' s imitation game : a discussion with the benefit of hindsight. In: . [S.l.: s.n.], 2004. Citado na página 19.

RAO, A. S. Agentspeak(l): Bdi agents speak out in a logical computable language. In: **Proceedings of the 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World: Agents Breaking Away: Agents Breaking Away**. Berlin, Heidelberg: Springer-Verlag, 1996. (MAAMAW '96), p. 42–55. ISBN 3540608524. Citado na página 24.

REDMON, J. et al. You only look once: Unified, real-time object detection. **CoRR**, abs/1506.02640, 2015. Disponível em: <http://arxiv.org/abs/1506.02640>. Citado na página 53.

REYS, A. et al. Predicting multiple icd-10 codes from brazilian-portuguese clinical notes. 07 2020. Citado na página 26.

RUSSELL, S.; NORVIG, P. **Artificial Intelligence: A Modern Approach**. 3. ed. [S.l.]: Prentice Hall, 2010. Citado 2 vezes nas páginas 9 e 24.

SARKER, I. Deep cybersecurity: A comprehensive overview from neural network and deep learning perspective. **SN Computer Science**, v. 2, 05 2021. Citado na página 27.

SARKER, I. Machine learning: Algorithms, real-world applications and research directions. **SN Computer Science**, v. 2, 05 2021. Citado 2 vezes nas páginas 25 e 26.

SILVER, D. et al. Mastering the game of go with deep neural networks and tree search. **Nature**, v. 529, p. 484–503, 2016. Disponível em: <http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html>. Citado 2 vezes nas páginas 30 e 31.

STANLEY, K. O.; MIIKKULAINEN, R. Evolving neural networks through augmenting topologies. **Evolutionary Computation**, v. 10, n. 2, p. 99–127, 2002. Citado na página 54.

SUTTON, R. S.; BARTO, A. G. **Reinforcement Learning: An Introduction**. Second. The MIT Press, 2018. Disponível em: <http://incompleteideas.net/book/the-book-2nd.html>. Citado na página 27.

TSAY, J.-J.; CHEN, C.-C.; HSU, J.-J. Evolving intelligent mario controller by reinforcement learning. In: **2011 International Conference on Technologies and Applications of Artificial Intelligence**. [S.l.: s.n.], 2011. p. 266–272. Citado 2 vezes nas páginas 30 e 31.

TURING, A. M. I.—COMPUTING MACHINERY AND INTELLIGENCE. **Mind**, LIX, n. 236, p. 433–460, 10 1950. ISSN 0026-4423. Disponível em: <https://doi.org/10.1093/mind/LIX.236.433>. Citado na página 19.

ZATELLI, M. et al. Smadas: A team for mapc considering the organization and the environment as first-class abstractions. In: . [S.l.: s.n.], 2013. v. 8245, p. 319–328. Citado 2 vezes nas páginas 29 e 31.

Zatelli, M. R. et al. A proposal of qlearning to control the attack of a 2d robot soccer simulation team. In: **2012 Brazilian Robotics Symposium and Latin American Robotics Symposium**. [S.l.: s.n.], 2012. p. 174–178. Citado 3 vezes nas páginas 19, 29 e 31.

ZHU, W. et al. GMAIR: unsupervised object detection based on spatial attention and gaussian mixture. **CoRR**, abs/2106.01722, 2021. Disponível em: <https://arxiv.org/abs/2106.01722>. Citado na página 52.

Apêndices

APÊNDICE A – Artigo

Neste apêndice será apresentado o artigo no formato SBC, referente ao presente projeto.

KITSUNE AI: UMA ARQUITETURA UTILIZANDO AGENTE BDI E APRENDIZAGEM POR REFORÇO PARA JOGAR JOGOS DE NES.

Robson Zagre Júnior [1]

[1]Departamento de Informática e Estatística – Universidade Federal de Santa Catarina (UFSC)
Florianópolis – SC – Brasil

r.zagre.jr@gmail.com

Abstract. *This paper creates a modular architecture that allows the development of an Agent that detects objects on the screen to be analyzed by the Belief–Desire–Intention (BDI) approach, supported by Reinforcement Learning algorithms, to design plans with the objective of play NES games, in this case Super Mario Bros. In the implementation, the agent is able to detect obstacles, enemies and other objects on the screen used to create perceptions, defining the next action. QLearning and SARSA algorithms were used, demonstrating the modular architecture that allows targeted experiments and improvements.*

Resumo. *Esse trabalho cria uma arquitetura modular que permite o desenvolvimento de um Agente que detecte objetos na tela para serem analisados pela abordagem Belief–Desire–Intention (BDI) amparada com algoritmos de Aprendizado por Reforço na concepção de planos com o objetivo de jogar jogos do NES, nesse caso Super Mario Bros. Na implementação, o agente é capaz de detectar obstáculos, inimigos e outros objetos da tela utilizados na criação de percepções, definindo a próxima ação por meio. Foram utilizados os algoritmos QLearning e SARSA, demonstrando a arquitetura modular que permite experimentos e melhorias direcionados.*

1. Introdução

Com o constante avanço nas diferentes partes da IA, algoritmos de aprendizado de máquinas e de agentes estão sendo aprimorados, permitindo uma melhoria nos resultados [Krizhevsky et al. 2012]. Por mais que ambas as abordagens estejam voltadas para a solução de problemas aparentemente antagônicos, a utilização delas em conjunto pode ser benéfica para determinados problemas, como a execução da melhor ação por um robô jogando futebol [Zatelli et al. 2012].

Algoritmos inteligentes capazes de tomar decisões em um ambiente de forma autônoma, levando em consideração seus conhecimentos e raciocínios, são chamado de Agentes. O modelo *Belief–Desire–Intention* (BDI) [Bratman 1987] é um exemplo de agente no qual, por meio de um ciclo de raciocínio, há o desenvolvimento de planos com o objetivo de alcançar uma meta definida. Por outro lado, possuímos algoritmos que detectam padrões e informações em dados observados, conseguindo prever um possível resultado no futuro com base em modelos estatísticos e funções matemáticas, chamados de Aprendizado de Máquinas. A técnica de Relearning, por exemplo, demonstra a capacidade de adaptabilidade e aprendizagem de acordo com os dados [Barto 1998].

Um ambiente de jogos eletrônicos é um espaço virtual que apresenta, normalmente, um objeto controlado pelo jogador que efetua alguma ação com o intuito de cumprir um objetivo. Isso cria um cenário que permite testar ferramentas e estratégias de implementações de agentes autônomos capazes de se adaptar e aprender com o meio, além de facilitar a comparação [Krukoski 2019]. Esse trabalho analisa a união de Aprendizado por Reforço em Agentes BDI, utilizando o ambiente virtual do jogo *Super Mario Bros* pra avaliar a implementação de uma arquitetura de inteligência artificial híbrida capaz de jogar jogos de *Nintendo Entertainment System* (NES).

2. Fundamentação Teórica

2.1. Inteligência Artificial Híbrida

A área de Inteligência Artificial pode ser dividia em: IA simbólica e IA não simbólica, cada uma possuindo assim suas particularidades quanto a implementação e solução de problemas [Nilsson 2010]. Essas áreas possuem subdivisões internas relacionadas às especificações encontradas no desenvolvimento, tecnologia, problemática e afins. A utilização de diversas abordagens na construção de um sistema mais complexo vem sendo estudada como Inteligência Artificial Híbrida. A ideia basicamente se baseia na união de diferentes técnicas, visando um sistema composto por partes com focos específicos balanceando assim seus benefícios e limitações individuais.

2.2. Agentes

Agentes são entidades inteligentes que são programadas com o intuito de realizar interações com o ambiente. O *Belief Desire Intention* (BDI) é uma abordagem que visa a modelagem de agentes baseada em conceitos do raciocínio humano. Como o próprio nome sugere, a metodologia é fundamentada em crenças, desejos e intenções, ou seja, o programador coloca uma gama de conhecimento base como as crenças que o agente possui do mundo, que são flexíveis e podem sofrer alterações. Além disso, há os desejos que são os objetivos do agente, ou seja, as metas que devem ser cumpridas nesse cenário. Baseando-se nisso, a execução de suas intenções é expressada por meio dos planos de execução, os quais são previamente programados [Bratman 1987].

2.3. Aprendizado por Reforço

Aprendizado por Reforço é uma das técnicas de ML baseada em recompensas. Há um agente no ambiente recebendo informações, as quais são utilizadas e processadas na elaboração de uma resposta possível que é materializada em uma ação. Após isso, o agente percebe o efeito da ação no ambiente pela análise do retorno da função de recompensa, ponderando assim os pesos dos fatores de probabilidade de tal forma que permita um ajuste positivo com o avanço do treinamento [Barto 1998].

Abordagens clássicas de Aprendizado por Reforço são baseadas em tabelas de probabilidade, como os algoritmos de QLearning e SARSA [Sutton and Barto 2018]. Ambos são bem parecidos, como podemos observar pelas equações representadas nas Figuras 1 e 2. De forma simplificada, uma tabela é criada possuindo respectivamente como linhas e colunas os estados e ações possíveis do ambiente. Os valores dessa tabela representam as probabilidades ponderadas, de acordo com as equações, de uma ação ocorrer. Quando um passo de treino é efetuado para um estado s_t, analisa-se a recompensa r_t resultante da

ação a_t obtida por meio da *policy*, ponderando os valores do estado/ação de acordo com uma taxa de aprendizado α e uma taxa γ de importância das ações a futuras no estado s_{t+1}. Para o QLearning, pega-se sempre a melhor ação futura possível, já para o SARSA, a ação é definida de acordo com uma *policy*.

$$Q^{new}(s_t, a_t) \leftarrow \overbrace{\underbrace{Q(s_t, a_t)}_{oldvalue} + \underbrace{\alpha}_{learningrate} \cdot \Big(\underbrace{\underbrace{r_t}_{reward} + \underbrace{\gamma}_{discountfactor} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{estimateofoptimalfuturevalue}}_{newvalue(temporaldifferencetarget)} - \underbrace{Q(s_t, a_t)}_{oldvalue} \Big)}^{temporaldifference}$$

Figure 1. Equação de atualização dos valores no algoritmo QLearning. [1]

$$Q^{new}(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_t + \gamma\, Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right]$$

Figure 2. Equação de atualização dos valores no algoritmo SARSA [2]

Concomitantemente, há técnicas que possuem conceitos e metodologias diferentes da anterior, como é o caso dos Algoritmos de Evolução Genética, baseados na evolução natural dos seres vivos. A ideia principal é constituída por cadeias de DNA sofrendo cruzamentos dos indivíduos que obtiveram a melhor performance em um ciclo de vida. Assim, os genes são perpetuados para as próximas gerações, podendo inclusive sofrer mutação decorrente de um fator X que visa dificultar a estabilidade e aumentar a diversidade [Mitchell 1998].

2.4. Visão Computacional

O ramo de visão computacional tem como objetivo a obtenção de informações em imagens. Inicialmente, no que chamamos hoje de abordagem clássica, os estudos se baseavam na manipulação das imagens por meio de técnicas de detecção de contornos, segmentação, filtros e afins, que geravam dados utilizados em modelos de classificação. No século 21, a utilização de Deep Learning começou a ganhar espaço após o vencimento da AlexNet na competição de visão computacional [Krizhevsky et al. 2012]. A ideia se baseava na *Convolutional Neural Networks* (CNNs), permitindo assim, a abstração dos problemas e aumentado o dinamismo das soluções, as quais eram antes extremamente específicas. Abordagens híbridas podem ser interessantes por aproveitarem a simplificação do processamento com técnicas clássicas além da generalização disponibilizada pelos algoritmos novos [Mahony et al. 2019].

3. Trabalhos Relacionados

A implementação de Aprendizado de Máquinas visando otimizar a decisão de agentes no momento de efetuar a ação do chute na *RoboCup* [Krukoski 2019]. Durante o artigo, é feito uma comparação entre os times com e sem a utilização das redes neurais, demonstrando

[1] Latex da equação obtido em: `https://en.wikipedia.org/wiki/Q-learning#Algorithm`

[2] Latex da equação obtido em: `https://en.wikipedia.org/wiki/State-action-reward-state-action#Algorithm`

uma maior vantagem no primeiro caso. Um conceito parecido apresenta a aplicação do algoritmo de QLearning na implementação do ataque do time [Zatelli et al. 2012].

Além dessa abordagem, percebemos também o estudo voltado para a construção estrutural do agente [Bosello 2020] utilizando um framework chamado de **Jason-RL** [Bosello and Ricci 2020]. Nesse caso, a criação de agentes utiliza a linguagem Jason em conjunto com técnicas de *Reiforcement Learning* (RL) na elaboração de planos temporários, os quais podem determinar e ajustar planos visando o software 2.0, ou seja, o desenvolvimento de programa por meio de programas. A implementação, que foi o caso de teste do framework, focou na elaboração de um modelo de carro autônomo, sendo treinado primeiramente em um ambiente virtual passando posteriormente para um protótipo de carro construído. Os autores explicaram como a abordagem BDI ficou responsável pela elaboração do percurso como um todo, deixando o foco dos ajustes e da imprevisibilidade da pista para o RL. A proposta apresentada nesse trabalho elabora uma arquitetura que contêm o framework **Jason-RL** [Bosello and Ricci 2020] na construção de agentes voltados para jogos de *NES*, incluindo o gerenciamento do ambiente e da extração de informações.

Já em [Neto 2016], há um trabalho de Inteligência Artificial em múltiplos níveis. A arquitetura BDI ficou responsável pelo nível cognitivo, o algoritmo de A* ficou responsável pelo nível instintivo e a lógica fuzzy ficou responsável pelo nível reativo. Dessa forma, o agente desenvolvido deveria participar de um jogo virtual simulando 'capture a bandeira', desenvolvendo estratégias e interpretando variáveis de entrada e saída.

Olhando agora para experimentos voltados para o jogo *Super Mario Bros*, observamos que técnicas de Aprendizado por Reforço no estudo e análise de abordagens para o desenvolvimento de um agente capaz de aprender por meio do ambiente do jogo podem ser utilizadas [Tsay et al. 2011]. Não obstante, os processos de decisão de *Markov* visando entender as particularidades dos jogadores humanos por meio de técnicas de Aprendizado por Reforço Inverso, podem ser aplicados para transmitir tal conhecimento posteriormente para o agente [Lee et al. 2014].

Uma abordagem utilizando o BDI, que possui um ciclo de raciocínio, no controle geral do agente em um ambiente de video game permite uma maior pró atividade, aumentando a chance de atuação em diversos cenários de um mesmo contexto. Aliado a isso, técnicas de Aprendizado por Reforço atreladas ao agente o tornam capaz de aprender como a mecânica de um jogo funciona, adaptando-se melhor ao meio otimizando suas ações.

4. Desenvolvimento

4.1. Modelagem e Arquitetura

O conceito do projeto se baseia na elaboração de uma arquitetura que implemente um agente capaz de entender e se adaptar a um ambiente de video game. Assim, a aprendizagem sobre o meio e a elaboração de planos de acordo com seus objetivos são importantes. Como podemos observar na Figura 3, possuímos 3 grandes módulos para que tudo possa funcionar: um módulo dedicado ao ambiente do jogo, executando-o e obtendo informações específicas; um módulo dedicado a visão computacional responsável pela simplificação e extração do estado atual e por fim, um módulo de agente que entende, raciocina e gera a próxima ação do jogador de acordo com a metodologia BDI apoiada pelo RL.

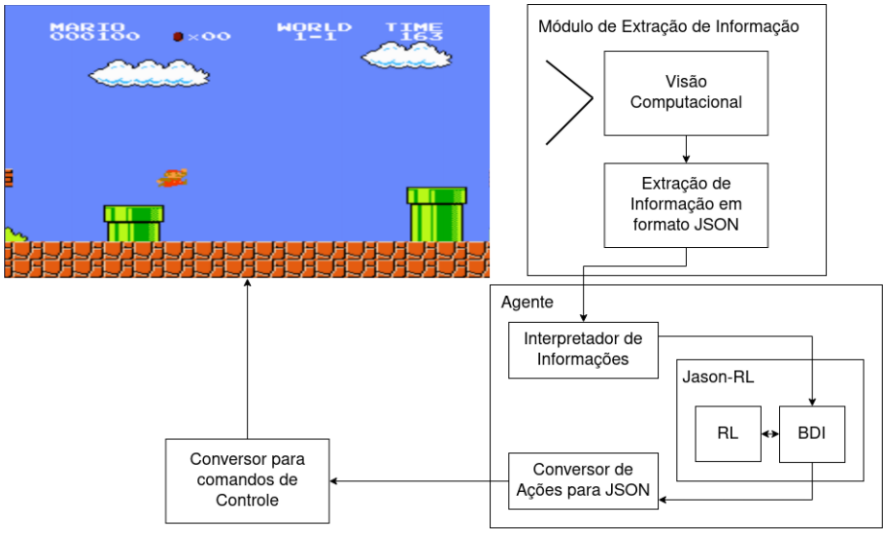

Figure 3. Arquitetura da implementação

4.2. Kitsune AI

A Programação Orientada a Objetos foi o paradigma escolhido para desenvolver o código. Ela permite uma melhor organização dos conceitos, necessário por conta da complexidade do projeto. Com isso, os módulos poderão ser facilmente aprimorados de forma independente, tornando o trabalho mais flexível. Dito isso, classes existem dentro da classe principal de controle (KitsuneAI), cada uma responsável pelo gerenciamento de um conceito demonstrado na seção 4.1: ambiente virtual (KitsuneEnv); visualização (KitsuneView) e agente (KitsuneAgent).

4.2.1. KitsuneEnv

Como comentado anteriormente, KitsuneEnv foi o nome escolhido para a classe responsável pelo gerenciamento do ambiente virtual e foi implementada utilizando o módulo Nes-py [Kauten 2018a]. Para tal, foi necessário o entendimento da renderização do jogo e do módulo *pyglet*[3] utilizado para controlar a interface gráfica. O módulo então foi construído possuindo a instância de outra classe responsável por definir algumas questões particulares para cada jogo escolhido. Nesse caso, para o jogo *Super Mario Bros*, o desenvolvimento se baseou na implementação de exemplo chamada de ***Super Mario Bros for OpenAI Gym***, que demonstra a manipulação da memória do jogo na obtenção de informações e na execução de ações como pular a tela inicial, sendo criado uma versão chamada de KitsuneSuperMarioBrosEnv [Kauten 2018b] .

Esse módulo gerencia a obtenção de algumas informações úteis do jogo, o controle de quadros por segundo, do inglês *Frames Per Second* (FPS), a leitura de informações do teclado e a interação com o jogo. Para isso, como podemos observar na Figura 4, uma classe específica dedicada a ROM(*Read-Only Memory*) de cada jogo, nesse caso a implementação KitsuneSuperMarioBrosEnv, deve ser desenvolvida ficando responsável pela obtenção de informações interessantes na construção do agente, como a quantidade

[3]Módulo python multiplataforma de desenvolvimento de jogos e aplicações visuais

atual de pontos, a movimentação do jogador, o estado final de uma fase e a imagem atual do jogo por exemplo. Outras métricas podem ser obtidas por meio de cálculos da posição de objetos efetuando comparações com os estados anteriores por exemplo, sendo necessário uma análise do que faz sentido para o jogo escolhido. Com isso, as informações podem ser utilizadas por um jogador humano por meio do teclado, no modo **manual**, ou pelo módulo KitsuneAgent apoiado pelo KitsuneView, no modo **automático**, para se definir uma ação a qual é traduzida pelo **nes-py** na execução da próxima etapa do jogo. A escolha do modo de jogo é definida quando se pressiona a tecla M e é definido o FPS por meio do escalonamento de uma função baseada no tempo.

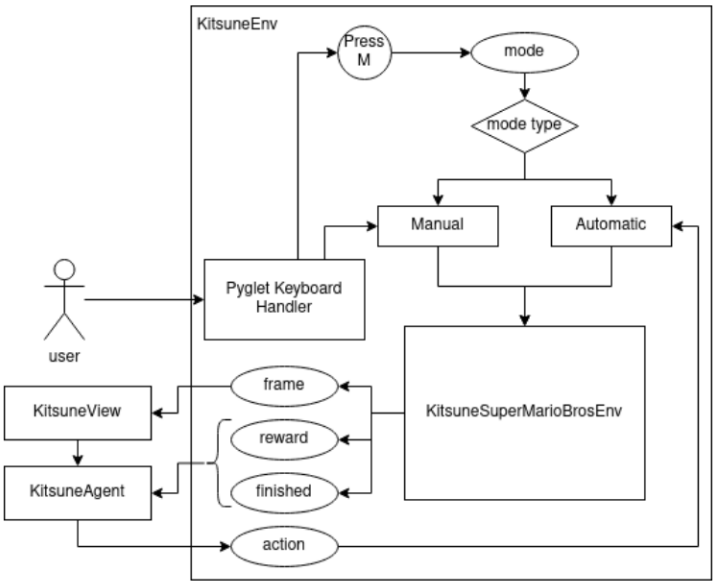

Figure 4. Visão geral do módulo de ambiente

4.2.2. KitsuneView

KitsuneView é a classe responsável pela extração de informações relevantes da tela do jogo para serem utilizadas pela IA. A abordagem adotada utiliza de métodos de Visão Computacional clássica, como apontado na seção 2.4. Baseando-se no trabalho de *Object Detection in Mario*, foram extraídos algumas imagens de elementos do jogo que foram utilizadas juntamente com o método de detecção de modelo do *opencv*[4] [Morarji 2020]. Para isso, durante a inicialização do módulo, convertemos todas as imagens de objetos do jogo para tons de cinza, carregando-as na memória do computador para serem posteriormente utilizadas, armazenando também outras informações. Para cada objeto que desejamos encontrar, a função **opencv.matchTemplate()** do módulo *opencv* é utilizada para procurar alguma ocorrência na imagem atual do jogo, também convertida em tons de cinza, retornando o grau de semelhança encontrada nos pixels. O retorno dessa função é um conjunto de pontos indicando onde esses padrões foram encontrados. Podemos assim utilizar as informações armazenadas anteriormente como dimensão, nome, e tipo para gerar um dicionario com dados indicando o estado atual do jogo. Para acelerar o processo, a repetição dessa etapa para cada imagem de objeto que possuímos é realizada de forma

[4]Biblioteca de tratamento e manipulação de imagens

paralela pelo módulo *multiprocessing*[5] do python. Além disso, de forma a simplificar algumas informações para o agente, pontos do mesmo objeto que estão localizados no mesmo local horizontalmente e que, verticalmente estejam contidos uns nos outros, são unidos a fim de se criar uma única informação.

Essa abordagem pode não ser a melhor quando pensamos em complexidade computacional, podendo possuir outros métodos e técnicas que possuam resultados satisfatórios com complexidades menores. Para entender melhor isso, para cada imagem de objeto I, executamos uma busca utilizando a função **opencv.matchTemplate()**, além de um for para cada ponto J encontrado buscando concatenar as informações. O método **opencv.matchTemplate()** utiliza do *Fast Fourier Transform*, que possui complexidade de $O(NlogN)$. Ou seja, de forma simplificada, a complexidade desse projeto é de $O(I \times J \times N \times logN)$, ou seja, $O(N^3logN)$.

4.3. KitsuneAgent

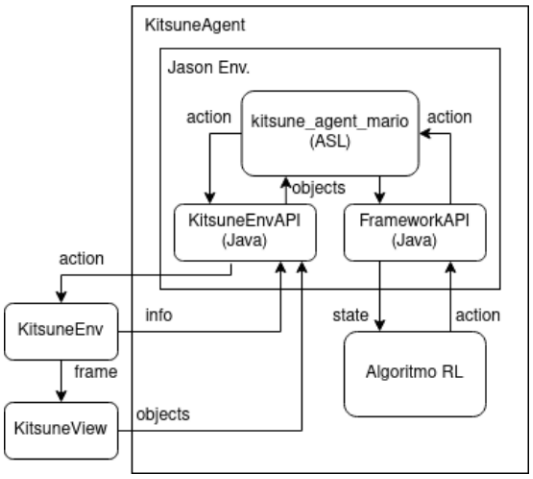

Figure 5. Visão geral da arquitetura do agente.

Essa classe é responsável pela integração do agente BDI com os algoritmos de RL por meio do framework **Jason-RL** [Bosello and Ricci 2020], bem como da utilização das classes citadas acima. Como podemos ver pela Figura 5, internamente essa classe possui uma referência tanto à KitsuneEnv quanto à KitsuneView instanciadas pela KitsuneAI. Quando um passo do processo ocorre, utilizando o módulo de ambiente, a KitsuneAgent pega as informações atuais do estado do jogo, resumindo-se na imagem atual, na recompensa e se é um estado final. Utilizando agora o módulo de visualização, são extraídos os objetos da cena, obtendo-se métricas adicionais pela classe do jogo que podem ser interessantes para o agente. No caso escolhido, as informações de velocidade dos objetos são importantes por demonstrarem, por exemplo, objetos relacionados ao cenário além de objetos que se movem, bem como o estado atual do Mario, se o mesmo está preso ou está se locomovendo, sendo obtidas realizando-se um cálculo simples comparando as posições dos objetos do estado atual com o estado anterior. Com essas informações, o processo segue por meio das comunicações da *Application Programming Interface*(API) com o cenário desenvolvido em Java que passará os objetos para o Cartago, podendo assim ser

[5]Módulo python que disponibiliza ferramentas de programação paralela e concorrente

observado pelo Agente Jason, que utilizará novamente da API para obter as respostas da implementação de RL que resultará na próxima ação do jogo.

4.3.1. Jason

O desenvolvimento do agente Jason acaba sendo específico para o jogo escolhido. No caso desse trabalho, para o *Super Mario Bros*, algumas regras e ações foram desenvolvidas para que o agente consiga detectar informações relevantes do meio. Independente da utilização, uma questão importante que deve ser levada em consideração na hora de tomar decisões nesse jogo é descobrir no que o jogador está encostando. Logo, um conjunto de regras comparando as posições X e Y foram desenvolvidas, retornando a posição que os objeto 1 está tocando no objeto 2, no caso ***top, right, bottom, left***. Algumas outras regras foram criadas utilizando essas como base para verificarem especificamente se o jogador está tocando algo e o local.

Utilizando essa informação, juntamente com outras condicionas, como por exemplo se é gameover, foram desenvolvidos ações que classificam determinados tipos de objetos em **"inimigos"** ou **"caminhos"**. Ou seja, adicionamos a crença de que um determinado tipo de objeto pode estar atrelado com certas características. Com isso, pensamos que ao passar o conjunto de crenças para os algoritmos de RL, uma relação de que objetos do tipo "inimigo" estão relacionados com a perda no jogo, e que por causa disso, qualquer objeto classificado com essa etiqueta deve ser evitado. A mesma ideia se aplica para os objetos os quais eu posso caminhar sobre, permitindo assim que eu prossiga pelo jogo.

4.3.2. Algoritmo de Aprendizado por Reforço

O framework **Jason-RL** apresenta a possibilidade de implementação de algoritmos de RL tanto em java quanto em python direcionados a biblioteca TensorFlow [Bosello and Ricci 2020]. Dessa forma, o conjunto de estados é criado pensando nessa utilização, sendo necessário o entendimento das transformações de estado para que os mesmos sejam transferidos para qualquer implementação de RL. Após efetuar a conexão por meio da API, verificou-se que o estado que chegava não condizia com toda a visualização encontrada pelo agente, sendo necessário efetuar uma mudança no framework para obter todas as percepções do agente. Atrelado a isso, modificou-se a obtenção dos parâmetros de cada percepção para que valores nulos sejam aceitos além da criação das ações possíveis no framework pois, como era baseado em um Set e não em uma Lista, a cada inicialização da aplicação os valores numéricos referentes as ações variavam.

Para validação da arquitetura e das modificações necessárias, foram desenvolvidos, em python, os algoritmos de QLearning e SARSA explicados na seção 2.3, sendo implementado a $policy$ ϵ-$greedy$ a qual apresenta uma probabilidade ϵ de se ocorrer uma ação aleatória, escolhendo-se a melhor ação caso contrário. Atrelado a isso, o ϵ sofre um decaimento a cada passo, sendo seu valor multiplicado pelo ϵ-$decay$.

Na implementação, o estado primeiro é convertido de listas para tuplas, para permitir a utilização como chaves de dicionários, representando nossas tabelas, sendo posteriormente ordenados a fim de trazer o padrão de que tipos específicos de observações

virão na frente de outras. Por fim conseguimos salvar esses valores por meio da biblioteca *pickle*[6] para garantir treinamento, utilização futura e observação de métricas.

4.3.3. Algoritmo de Evolução Genética

Embora o foco do trabalho seja Aprendizado por Reforço, de forma a verificar a flexibilidade do código desenvolvido, bem como outros possíveis resultados, foi implementado um algoritmo simples de Evolução Genética baseada em Redes Neurais [Mitchell 1998]. De forma simplificada um organismo possui uma primeira camada de neurônios com $(256 * 240)/(16 * 16) * 9$ entradas levando-se em consideração que as imagens dos objetos possuem, normalmente, 16 pixels de largura e altura, e que a tela do jogo é de 256x240 pixels, cada um desses objetos possuem 9 informações ao todo, resultando no número total teórico de informação que pode ocorrer. Após isso, a rede segue com 3 camadas de 16 neurônios e finaliza com uma camada de 12 neurônios demonstrando as opções de ações finais sendo ativados com *softmax* visando apenas uma ativação. Todas as camadas são totalmente conectadas. Uma população de 50 organismos foi escolhido para representar uma geração, possuindo uma taxa de 0.5 de mutação com decaimento de 0.9 a cada geração. Para gerenciar o estado recebido do JASON, que é em formato matricial, efetuamos simplesmente uma ordenação seguido de redução para 1 dimensão.

Quando o código começa a ser executado, uma nova população é desenvolvida e o primeiro organismo é eleito para decidir as ações no ambiente. Se a cada 100 decisões não houver uma melhoria na recompensa acumulada, então força-se um reset do sistema para ser avaliado o próximo organismo. Quando todos os membros da geração são avaliados, pega-se os 10% que melhor pontuaram e cruza-os com os demais indivíduos seguindo uma probabilidade de escolha baseada no desempenho, criando-se assim a próxima geração. Atrelado a isso, cada ”DNA”, no caso representado pelas redes neurais, possui uma probabilidade de sofrer mutação seguindo um desvio do valor herdado pelos pais. Por fim o módulo *dill*[7] é utilizado para salvar os treinos, o que permite a comparação, análise e reprodutibilidade futura.

5. Resultados e Discussões

Foi desenvolvido um agente capaz de jogar *Super Mario Bros* que identifica os conceitos de ”inimigo” e ”caminho”. Com isso, conseguimos atrelar a um agente, que aprendeu a se locomover no meio utilizando técnicas de aprendizado por reforço e evolução genética, conceitos simbólicos e de ”alto” nível os quais nós humanos normalmente desenvolvemos ao jogar esses tipos de jogos. A arquitetura implementada permite uma fácil troca entre os algoritmos de RL, permitindo a comparação entre diferentes técnicas como foi ilustrado pelo QLearning e SARSA, sendo possível inclusive o estudo de outras abordagens de Aprendizado de Máquinas, como demonstrado pela implementação do algoritmo de Evolução Genética com Redes Neurais.

Algoritmos clássicos de RL não apresentam grandes resultados em cenários complexos como é o caso do *Super Mario Bros*. Mesmo efetuando o treinamento por mais de 430 episódios, não percebemos um aumento significativo na recompensa acumulada. Há

[6]Módulo python de serialização e deserialização binária.
[7]Módulo python de serialização e deserialização binária.

uma convergência para valores próximos a 100, não alcançando por exemplo valores significativamente maiores. Esse resultado pode ter sido por conta dos parâmetros escolhidos: alpha = 0.75; gamma = 0.8; epsilon = 0.9; epsilon_decay = 0.95; epsilon_min = 0, os quais foram escolhidos de forma arbitrária seguindo simplesmente o significado dos parâmetros, sem efetuar uma otimização aprofundada. Mesmo assim, isso não resolve o problema da falta de aproveitamento de conhecimento de estados parecidos, o que as implementações utilizadas não levam em consideração. Isso pode ter dificultado a convergência do algoritmo em decorrência da grande quantidade de estados possíveis juntamente com o número significativo de ações possíveis.

Algoritmos que possuem funções sendo ponderadas de acordo com os valores encontrados conseguem criar relações entre os números apresentados de forma mais interessante do que as abordagens clássicas de RL. No caso do algoritmo de Evolução Genética de uma Rede Neural, percebemos durante o treinamento de 52 gerações uma tendência crescente, possuindo picos acima de 600 em alguns casos. Vale ressaltar aqui que cada geração possui 50 organismos analisados, e que alterando algumas questões como taxa de mutação, decaimento da mutação, número da população, método de cruzamento e até mesmo a topologia da rede, resultados melhores poderiam ser obtidos visto que os valores também foram escolhidos de forma arbitrária. Diferentemente das abordagens anteriores, cada ação definida pelo agente é executada por 3 passos consecutivas, simplificando a quantidade de informação recebida.

Algumas técnicas de detecção de objetos foram rapidamente experimentadas na elaboração desse projeto. O algoritmo *Self-Supervised DINO* acabou necessitando de muita memoria para execução [Caron et al. 2021]. Já outras abordagens não supervisionadas como *SPAIR* [Crawford and Pineau 2019], *SILOT* [Crawford and Pineau 2020], *SPACE* [Lin et al. 2020] e *GMAIR* [Zhu et al. 2021], por mais que possuem um bom desenvolvimento em jogos de Atari, o mesmo não ocorreu em cenários mais complexos como o do *Super Mario Bros*.

6. Conclusão

O presente trabalho apresentou informações relacionadas a diferentes frentes da área de Inteligência Artificial, servindo como referência para o entendimento dessas áreas e suas aplicações, bem como do desenvolvimento de uma arquitetura híbrida e modularizada que permite a criação de um agente capaz de aprender a jogar jogos de NES. Como podemos observar na seção 5, algumas abordagens na área da visão computacional foram experimentadas, demonstrando que podemos alterar essa parte da implementação sem impactar significativamente os demais módulos. Além disso, podemos expandir os algoritmos de RL que podem ser instanciados pela KitsuneAgent. Dito isso, foram pesquisados as áreas de Aprendizado por Máquinas bem como algoritmos de Aprendizado por Reforço e Agentes BDI, entendendo arquiteturas e modelagens existentes nesses casos. Após levantamento do estado da arte em AI híbrida, analisando projetos que apresentam diferentes objetivos e metodologias, elaborou-se um modelo e uma arquitetura que representa as grandes divisões e responsabilidades necessárias. Com isso, foi desenvolvido um agente BDI juntamento com algoritmo de RL ou Evolução Genética que permite o aprendizado do jogo do Mário, o qual consegue detectar obstáculos, inimigos, caminhos e outros objetos na tela. A implementação e a modularização da proposta se demonstrou escalável e de fácil alteração como demonstrado pela implementação de outra técnica de aprendizado. O

código fonte pode ser encontrado no repositório do github Kitsune_AI[8].

Analisando as abordagens de Aprendizado por Reforço clássico e Evolução Genética com Redes Neurais, percebemos um melhor desempenho na última. Isso indica que algoritmos mais modernos tendem a entender melhor as correlações dos valores extraídos pela técnica BDI e de visão computacional, podendo inclusive tornar o aprendizado relativamente genérico dentro do contexto do jogo, o qual pode ser validado efetuando-se um treinamento em uma fase e o teste em outra.

7. Trabalhos Futuros

Futuramente, algumas técnicas de Visão Computacional podem ser implementadas e avaliadas para a detecção de objetos do jogo. Algoritmos supervisionados podem trazer bons resultados e com um desempenho melhor do que o obtido, visto que há arquiteturas voltadas à detecções em tempo real, como por exemplo a YOLO [Redmon et al. 2015].

Além disso, não foram explorados muitos algoritmos de aprendizado por reforço, podendo ser aplicadas técnicas mais sofisticadas que agregam redes neurais, utilizando-se da arquitetura desenvolvida para identificar a melhor técnica para o cenário do jogo. Não obstante, melhores técnicas de Evolução Genética podem ser testadas, como por exemplo evoluções de topologia de rede [Stanley and Miikkulainen 2002]. Uma ampliação da arquitetura para múltiplas execuções simultâneas pode ser implementada, o que se resumiria em várias instâncias da classe KitsuneAI.

Por fim, a arquitetura pode ser utilizada em outro jogo de NES a fim de verificar sua flexibilidade e modularização. O intuito da arquitetura é facilitar a modificação de cada módulo para reduzir as configurações necessárias nos demais, respeitando, obviamente, as implementações específicas existentes, como é o caso dos planos do agente Jason.

References

Barto, R. S. S. A. G. (1998). Adaptive computation and machine learning. In *Adaptive computation and machine learning*. Cambridge, Mass. : MIT Press, ©1998.

Bosello, M. (2020). Integrating bdi and reinforcement learning: the case study of autonomous driving. Master's thesis, University of Bologna.

Bosello, M. and Ricci, A. (2020). From programming agents to educating agents – a jason-based framework for integrating learning in the development of cognitive agents. In Dennis, L. A., Bordini, R. H., and Lespérance, Y., editors, *Engineering Multi-Agent Systems*, pages 175–194, Cham. Springer International Publishing.

Bratman, M. (1987). *Intention, Plans, and Practical Reason*. Cambridge: Cambridge, MA: Harvard University Press.

Caron, M., Touvron, H., Misra, I., Jégou, H., Mairal, J., Bojanowski, P., and Joulin, A. (2021). Emerging properties in self-supervised vision transformers. *CoRR*, abs/2104.14294.

Crawford, E. and Pineau, J. (2019). Spatiial invariant unsupervised object detection with convolutional neural networks. In *Thirty-Third AAAI Conference on Artificial Intelligence*.

[8]Github: `https://github.com/robsonzagrejr/kitsune_ai`

Crawford, E. and Pineau, J. (2020). Exploiting spatial invariance for scalable unsupervised object tracking. In *Thirty-Fourth AAAI Conference on Artificial Intelligence*.

Kauten, C. (2018a). Nes-py. GitHub.

Kauten, C. (2018b). Super Mario Bros for OpenAI Gym. GitHub.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Pereira, F., Burges, C., Bottou, L., and Weinberger, K., editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc.

Krukoski, F. M. (2019). ImplementaÇÃo e anÁlise de diferentesalgoritmos de aprendizado de mÁquina noambiente 2d simulado da robocup.

Lee, G., Luo, M., Zambetta, F., and Li, X. (2014). Learning a super mario controller from examples of human play. In *2014 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8.

Lin, Z., Wu, Y.-F., Peri, S. V., Sun, W., Singh, G., Deng, F., Jiang, J., and Ahn, S. (2020). Space: Unsupervised object-oriented scene representation via spatial attention and decomposition. In *International Conference on Learning Representations*.

Mahony, N. O., Campbell, S., Carvalho, A., Harapanahalli, S., Velasco-Hernández, G. A., Krpalkova, L., Riordan, D., and Walsh, J. (2019). Deep learning vs. traditional computer vision. *CoRR*, abs/1910.13796.

Mitchell, M. (1998). *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA.

Morarji, V. (2020). Object detection in mario. GitHub.

Neto, J. R. V. (2016). ComposiÇÃo de tÉcnicas de inteligÊncia artificial em uma arquitetura multi nível para emergÊncia de comportamentos em agentes cooperativos.

Nilsson, N. (2010). *The quest for artificial intelligence: A history of ideas and achievements*.

Redmon, J., Divvala, S. K., Girshick, R. B., and Farhadi, A. (2015). You only look once: Unified, real-time object detection. *CoRR*, abs/1506.02640.

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

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. The MIT Press, second edition.

Tsay, J.-J., Chen, C.-C., and Hsu, J.-J. (2011). Evolving intelligent mario controller by reinforcement learning. In *2011 International Conference on Technologies and Applications of Artificial Intelligence*, pages 266–272.

Zatelli, M. R., Neri, J. R. F., d. Santos, C. H. F., and Fabro, J. A. (2012). A proposal of qlearning to control the attack of a 2d robot soccer simulation team. In *2012 Brazilian Robotics Symposium and Latin American Robotics Symposium*, pages 174–178.

Zhu, W., Shen, Y., Yu, L., and Sanchez, L. P. A. (2021). GMAIR: unsupervised object detection based on spatial attention and gaussian mixture. *CoRR*, abs/2106.01722.