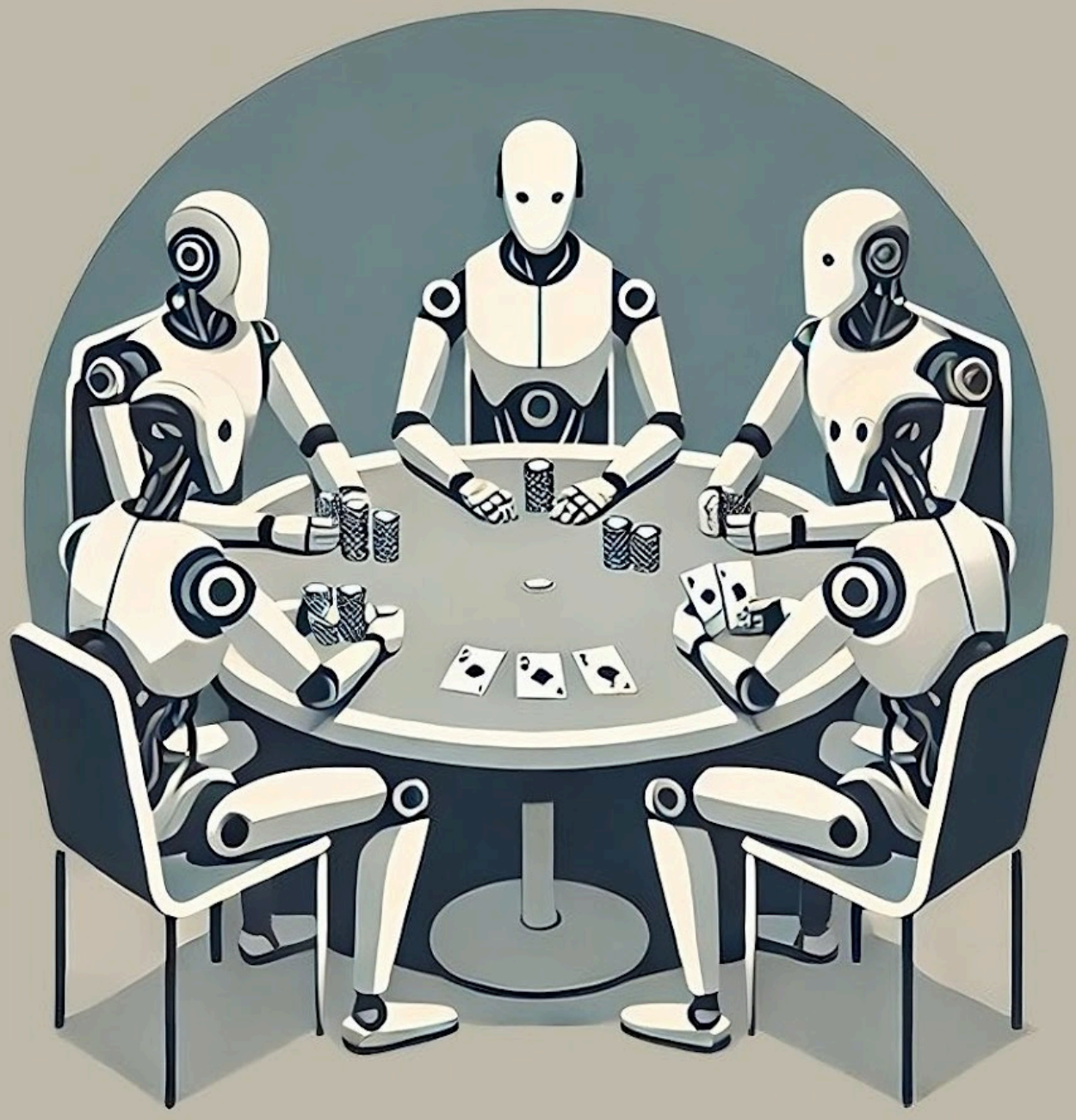
UNIVERSIDADE FEDERAL DE GOIÁS / INSTITUTO DE INFORMÁTICA

Aprendizado por Reforço em Jogos Multiagentes

Uma abordagem com Autojogo



André Luís Araújo de Souza

UFG
UNIVERSIDADE FEDERAL DE GOIÁS

UNIVERSIDADE FEDERAL DE GOIÁS (UFG)
INSTITUTO DE INFORMÁTICA (INF)

ANDRÉ LUÍS ARAÚJO DE SOUZA

Aprendizado por Reforço em Jogos Multiagentes

Uma abordagem com Autojogo

Goiânia
2025

UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

TERMO DE CIÊNCIA E DE AUTORIZAÇÃO PARA DISPONIBILIZAR VERSÕES ELETRÔNICAS DE TRABALHO DE CONCLUSÃO DE CURSO DE GRADUAÇÃO NO REPOSITÓRIO INSTITUCIONAL DA UFG

Na qualidade de titular dos direitos de autor, autorizo a Universidade Federal de Goiás (UFG) a disponibilizar, gratuitamente, por meio do Repositório Institucional (RI/UFG), regulamentado pela Resolução CEPEC no 1240/2014, sem ressarcimento dos direitos autorais, de acordo com a Lei no 9.610/98, o documento conforme permissões assinaladas abaixo, para fins de leitura, impressão e/ou download, a título de divulgação da produção científica brasileira, a partir desta data.

O conteúdo dos Trabalhos de Conclusão dos Cursos de Graduação disponibilizado no RI/UFG é de responsabilidade exclusiva dos autores. Ao encaminhar(em) o produto final, o(s) autor(a)(es)(as) e o(a) orientador(a) firmam o compromisso de que o trabalho não contém nenhuma violação de quaisquer direitos autorais ou outro direito de terceiros.

1. Identificação do Trabalho de Conclusão de Curso de Graduação (TCCG)

Nome(s) completo(s) do(a)(s) autor(a)(es)(as): ANDRÉ LUÍS ARAÚJO DE SOUZA

Título do trabalho: Aprendizado por Reforço em Jogos Multiagentes

Uma abordagem com Autojogo

2. Informações de acesso ao documento (este campo deve ser preenchido pelo orientador) Concorda com a liberação total do documento [X] SIM [ ] NÃO[1]

[1] Neste caso o documento será embargado por até um ano a partir da data de defesa. Após esse período, a possível disponibilização ocorrerá apenas mediante: a) consulta ao(à)(s) autor(a)(es)(as) e ao(à) orientador(a); b) novo Termo de Ciência e de Autorização (TECA) assinado e inserido no arquivo do TCCG. O documento não será disponibilizado durante o período de embargo.

Casos de embargo:

- Solicitação de registro de patente;

- Submissão de artigo em revista científica;

- Publicação como capítulo de livro.

Obs.: Este termo deve ser assinado no SEI pelo orientador e pelo autor.

Documento assinado eletronicamente por **André Luis Araújo De Souza**, **Discente**, em 13/01/2025, às 12:15, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Fernando Marques Federson**, **Professor do Magistério Superior**, em 15/01/2025, às 16:07, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020 .

A autenticidade deste documento pode ser conferida no site https://sei.ufg.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **5089458** e o código CRC **0AD2CDF5**.

Referência: Processo nº 23070.001541/2025-93 SEI nº 5089458

ANDRÉ LUÍS ARAÚJO DE SOUZA

Aprendizado por Reforço em Jogos Multiagentes

Uma abordagem com Autojogo

Relatório final de Trabalho de Conclusão de Curso, apresentado à Universidade Federal de Goiás, como parte das exigências para a obtenção do título de Bacharel em Inteligência Artificial.
Orientador: Prof. Dr. Fernando Marques Federson

Goiânia
2025

Ficha de identificação da obra elaborada pelo autor, através do
Programa de Geração Automática do Sistema de Bibliotecas da UFG.

SOUZA, ANDRÉ LUÍS ARAÚJO DE
Aprendizado por Reforço em Jogos Multiagentes [manuscrito] : Uma abordagem com Autojogo / ANDRÉ LUÍS ARAÚJO DE SOUZA. - 2025.
76 f.

Orientador: Prof. Dr. Fernando Marques Federson.
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Goiás, Instituto de Informática (INF), Inteligência Artificial, Goiânia, 2025.

1. inteligência artificial. 2. aprendizado por reforço. 3. sistemas multiagentes. I. Federson, Fernando Marques , orient. II. Título.

CDU 004

ANDRÉ LUÍS ARAÚJO DE SOUZA

Aprendizado por Reforço em Jogos Multiagentes

Uma abordagem com Autojogo

Relatório final de Trabalho de Conclusão de Curso, apresentado à Universidade Federal de Goiás, como parte das exigências para a obtenção do título de Bacharel em Inteligência Artificial.

Data da Aprovação: 17 de dezembro de 2024.

Prof. Dr. Fernando Marques Federson
Orientador (INF-UFG)

Prof. Dr. Aldo André Díaz Salazar
Coordenador de TCC do BIA (INF-UFG)

Prof. Dr. Anderson da Silva Soares
Coordenador do BIA (INF-UFG)

Me. Bruno Brandão Soares Martins
(CEIA-UFG)

ANDRÉ LUÍS ARAÚJO DE SOUZA

Aprendizado por Reforço em Jogos Multiagentes

Uma abordagem com Autojogo

RESUMO

Este Relatório de Conclusão de Curso tem como objetivo reunir os resultados da minha jornada para me tornar um especialista em **Aprendizado por Reforço (Multiagente)**. Uma ilustração e sua narrativa descrevem os períodos de trabalho. Os Apêndices contêm os Termos de Aceite de Entrega e os resultados obtidos durante cada período de trabalho.

Palavras-chave: inteligência artificial, modelos grandes de linguagem, geração automática de datasets.

ABSTRACT

This Course Completion Report aims to bring together the results of my journey to become an expert in **Reinforcement Learning (Multi-Agent)**. An illustration and its narrative describe the work periods. The Appendices contain the Delivery Acceptance Terms and the results obtained during each work period.

Keywords: artificial intelligence, large language models, automatic dataset generation.

Goiânia

2025

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

Minha Jornada

André Luís Araújo de Souza

Especialista em: Aprendizado por Reforço (Multiagente)

Instituto de Informática (INF) - UFG
Alameda Palmeiras, Quadra D, Câmpus Samambaia
Fone: (62) 3521-1181 / Fax: (62) 3521-1182

MINHA JORNADA

Nome: André Luís Araújo de Souza

Especialidade: Aprendizado por Reforço (Multiagente)

Objetivo deste documento

Durante o processo da disciplina Residência em IA[1], foram gerados diversos resultados na construção da minha especialização. A cada semana, um conjunto de resultados foi formalizado por um Termo de Aceite de Entrega e avaliado por uma banca, considerando o planejado e o realizado para o período. Este documento tem como objetivo descrever esses resultados obtidos, fazendo referência aos Termos de Aceite de Entrega e seus documentos associados.

Minha Jornada

Minha jornada começou na **Semana 1**, com a definição de uma área de especialização e o estudo de sua história. Escolhi Aprendizado por Reforço, um campo que despertou meu interesse durante o Bacharelado, graças à experiência em projetos de Pesquisa e Desenvolvimento e à disciplina homônima que cursei. Com a especialização definida, revisitei conceitos fundamentais e aprofundei-me na trajetória histórica desse campo. Como principal referência, utilizei o livro-texto *"Reinforcement Learning: An Introduction"*, de Barto e Sutton. A obra apresenta uma visão abrangente sobre o Aprendizado por Reforço, abordando suas principais técnicas e os desafios que elas ajudam a resolver. Após a leitura do primeiro capítulo, escrevi um resumo sobre a evolução histórica da área, consolidando os aprendizados iniciais. Além disso, revisei o material introdutório da disciplina de Aprendizado por Reforço cursada durante o Bacharelado. Nesse material, são apresentados conceitos como a função de valor e a Equação de Bellman, ferramentas

[1] Dez semanas, entre setembro de 2024 e dezembro de 2024.

essenciais para resolver problemas nesse campo. O resumo produzido e o material revisado foram incluídos no **Apêndice 1**.

Nas **Semanas 2 e 3**, dediquei-me a uma série de atividades para aprofundar meu conhecimento em Aprendizado por Reforço (RL) e organizar informações relevantes sobre o tema. Na **Semana 2**, realizei uma pesquisa de artigos fundamentais na área, compilando-os em uma lista estruturada, e iniciei a escrita de um texto com os conceitos fundamentais de RL. Também explorei hands-on os frameworks mais populares, como Gymnasium, Stable Baselines e Ray RLlib, cujos experimentos foram documentados em notebooks específicos, que podem ser encontrados no **Apêndice 2**. Para sintetizar as informações, elaborei um mapa mental simples representando as ramificações da área, com o objetivo de ter uma visualização mais clara de possíveis trilhas a serem seguidas, que também pode ser encontrado no **Apêndice 2**. Na **Semana 3**, aprofundei-me na leitura de artigos relevantes, como Deep Reinforcement Learning: A Survey (Wang et al., 2022) e Deep Reinforcement Learning: A Brief Survey (Arulkumaran et al., 2017), e atualizei tanto o documento de fundamentos quanto a lista de artigos utilizando a ferramenta Rabbit Research. Também participei de discussões com o grupo de estudo em RL do CEIA (Centro de Excelência em Inteligência Artificial - UFG) e iniciei uma pesquisa direcionada ao Aprendizado por Reforço Multiagente, consolidando avanços em minha compreensão teórica e prática da área.

Na **Semana 4**, as atividades foram focadas em aprofundar o estudo do Aprendizado por Reforço Multiagente (MARL) e avançar na definição de um ambiente para experimentação. Foram lidos os artigos *A Comprehensive Survey of Multiagent Reinforcement Learning* (Buşoniu et al., 2008) e *Multi-Agent Reinforcement Learning: A Review of Challenges and Applications* (Canese et al., 2021), cujos principais pontos foram sintetizados em um resumo dedicado, que pode ser acessado dentro do **Apêndice 3**. Além disso, foram consultados outros artigos relevantes, como *Multi-agent deep reinforcement learning: a survey* (Gronauer & Dielpold, 2021) e *The Complexity of Decentralized Control of Markov Decision Processes* (Bernstein et al., 2002), complementando a revisão bibliográfica. Também foi realizada uma pesquisa sobre frameworks para MARL, incluindo PettingZoo, MAgent2 e MARLlib, para identificar ferramentas adequadas para implementação prática. No campo teórico, avançou-se com a leitura do Capítulo 2 do livro *Reinforcement Learning:*

An Introduction. Por fim, foi esboçado um ambiente para experimentos, onde múltiplos agentes operam em um sistema financeiro com recursos limitados, interagindo por meio de ordens em pares de ativos e enfrentando desafios como custos operacionais periódicos e rendimentos decrescentes. A ideia era ter um ambiente misto (onde os agentes podem cooperar e/ou competir), para realizar estudos e experimentos,

Nas **Semanas 5 e 6**, o foco esteve no estudo de frameworks, através da implementação prática e da experimentação no contexto do Aprendizado por Reforço Multiagente (MARL). Na **Semana 5**, foi desenvolvida uma primeira versão de um simulador, que utiliza agentes puramente aleatórios para simular interações financeiras simples, seguindo os critérios definidos na semana anterior. Paralelamente, foram realizados experimentos com frameworks multiagentes, documentados no repositório MARL-Studies. No entanto, esses experimentos enfrentaram diversas dificuldades técnicas, resultando em um progresso limitado, com sucesso apenas em treinar uma iteração de um algoritmo em um ambiente monoagente. Também foram selecionados dois artigos para análise posterior, que fornecem insights importantes sobre simulação de agentes. Já na **Semana 6**, os experimentos com frameworks avançaram, com implementações bem-sucedidas nos ambientes *CartPole_v1* (pêndulo invertido), *tictactoe_v3* (ambiente adversarial simples) e *pistonball_v6* (ambiente cooperativo multiagente). Os códigos para esses experimentos podem ser encontrados no **Apêndice 4**. Além disso, foram lidos dois artigos relevantes, que levantaram questionamentos sobre a emergência de comportamentos específicos, como os de *market makers* e *liquidity takers*. Por fim, foi realizado um fork e a análise parcial do código do simulador *pymarketsim*, encontrado em um dos artigos, para explorar possíveis aplicações em experimentos futuros.

Nas **Semanas 7, 8 e 9**, as atividades foram marcadas por avanços técnicos e mudanças de direção para contornar desafios encontrados. Na **Semana 7**, foi configurado um ambiente reprodutível com Docker para facilitar os experimentos, corrigidos bugs relacionados à instanciação de bibliotecas e realizados testes iniciais com agentes de Zero Intelligence (ZI) no simulador PyMarketSim. Apesar desses progressos, surgiram dificuldades ao tentar implementar ambientes específicos para agentes *market makers* (MM), impedindo o avanço esperado nos treinamentos. Já na **Semana 8**, os esforços concentraram-se no debugging do

simulador PyMarketSim, com base em exemplos do repositório original. A análise revelou problemas críticos, como a criação de tensores de dimensão negativa devido a discrepâncias nos timestamps, e a necessidade de instanciar corretamente agentes MM para formatar o espaço de observações. Apesar de diversas tentativas de refatoração e investigação do código, a falta de conhecimento em Simulações de Mercado Financeiro apresentou um obstáculo significativo para compreender e ajustar a lógica subjacente. Por conta das dificuldades persistentes, a **Semana 9** marcou uma mudança estratégica para um ambiente de poker. Foram lidos e estudados o artigo *"Applying Reinforcement Learning to Poker"* e o código de referência associado, que já inclui políticas pré-implementadas (aleatória e baseada em “equity”). Os trabalhos iniciais envolveram ajustes e testes, seguidos pela implementação de um algoritmo DQN utilizando PyTorch, cuja evolução pode ser conferida nos commits recentes do projeto. Essa pivotagem abriu novas possibilidades para experimentos futuros. O link para os códigos desenvolvidos (simulador de mercado financeiro e ambiente de poker) pode ser encontrado no **Apêndice 5**.

Na **Semana 10**, o trabalho concentrou-se na implementação e análise de um experimento envolvendo treinamento *Self-play* em uma mesa de poker com três jogadores, bem como na avaliação do modelo treinado contra dois agentes aleatórios. Para isso, foi necessário adaptar o código do DQN desenvolvido anteriormente, separando o agente do módulo de treinamento, permitindo a atualização independente de múltiplos agentes durante o aprendizado. As alterações e aprimoramentos realizados estão documentados nos últimos commits do repositório *Neuron Poker*. Após corrigir alguns bugs, foram conduzidos dois experimentos principais: um com *Self-play* puro e outro onde o DQN enfrentava dois agentes aleatórios. Em seguida, o segundo experimento foi repetido por mais épocas para coleta de dados adicionais. As métricas e resultados obtidos foram registrados no TensorBoard e analisados por meio de gráficos disponíveis no repositório. A análise revelou desafios importantes, como possíveis erros no cálculo da recompensa para o *Self-play*, que geraram valores inconsistentes, e uma tendência geral de recompensas decrescentes ao longo dos episódios. Por outro lado, foi possível observar que o *Self-play* superava suas versões anteriores com consistência, apesar de os episódios frequentemente se alongarem devido a um comportamento de fuga dos agentes, algo também evidenciado nos testes, onde os índices de vitória foram extremamente baixos, atingindo 1% e 0%. Essa análise detalhada

foi consolidada em um documento específico, que pode ser encontrado no **Apêndice 6**. Por fim, destaquei dentro do Termo de Entrega da Semana, possíveis caminhos a serem seguidos a partir desse trabalho, como: aprimorar a função de recompensa, implementar um sistema de ligas para o *Self-play*, explorar abordagens como EGTA (*Empirical Game-Theoretic Analysis*), otimizar o treinamento com ambientes paralelos e testar algoritmos mais robustos, como A3C e PPO, visando corrigir os problemas observados e avançar no aprendizado.

Em função de tudo que vivi, posso dizer que minha Jornada foi extremamente enriquecedora, permitindo-me construir uma base sólida na área de Aprendizado por Reforço (Multiagente) e adquirir uma visão geral abrangente dos conceitos fundamentais. Além disso, desenvolvi experiência prática na implementação de algoritmos, tanto de forma independente quanto utilizando frameworks especializados, o que expandiu significativamente minhas habilidades técnicas. A oportunidade de elaborar e conduzir experimentos, bem como avaliar os resultados de forma crítica, foi particularmente transformadora, pois me deu a confiança de que posso contribuir academicamente para o avanço da área. Por fim, ao enfrentar e resolver problemas que se alinham ao escopo do Aprendizado por Reforço, amadureci minha capacidade de abordar desafios de maneira estruturada e criteriosa, consolidando um senso prático e analítico que será essencial em futuras contribuições científicas e aplicadas.

APÊNDICE 1

Termo de Aceite de Entrega

Objetivo deste documento

Este documento faz parte do Processo da disciplina Residência em IA e tem como objetivo formalizar o aceite da entrega considerando o planejado e o realizado para o período.

Data da Reunião ("gate") de aprovação: 19 de set. de 2024

Participantes da Entrega [matriculados em Residência em IA]:

ANDRÉ LUIS ARAÚJO DE SOUZA

Entrega: [descrever a ENTREGA: requisitos e produtos gerados: links para textos, códigos, vídeos etc.]

O trabalho dessa semana e meia foi dedicado ao estudo da história do **Reinforcement Learning**, área de especialização definida previamente. As atividades realizadas foram:

- Leitura do primeiro capítulo do livro Reinforcement Learning: An Introduction.pdf ;
- Visualização do vídeo The Story of Reinforcement Learning ;
- Escrita de um resumo sobre o conteúdo que foi consumido: Resumo História do RL .

Além da pesquisa sobre as origens do campo de estudo selecionado, também reproduzi um notebook de exercícios da disciplina cursada no sexto período, que aborda conceitos fundamentais envolvendo a função de valor: 01. Função de Valor.ipynb .

Planejamento: [descrever o que pretende fazer para realizar a próxima ENTREGA]

O planejamento para a próxima semana é:

- Pesquisar artigos fundamentais para a área de Aprendizado por Reforço;
- Compilar esses artigos em uma base bibliográfica;
- Elaborar uma visualização simples com o registro das sub-áreas de Aprendizado por Reforço;
- Pesquisar frameworks mais populares na área e fazer um hands-on;

Observação: [caso precise fazer alguma observação, de qualquer "natureza"]

ACEITE DA ENTREGA:

CEDRIC LUIZ DE CARVALHO: Go!

História do Aprendizado por Reforço

A história do Aprendizado por Reforço pode ser divida em três linhas temporais distintas que evoluíram independentemente umas das outras mas que culminaram nessa área de pesquisa a partir de um certo tempo.

Aprender por tentativa e erro

A noção do aprendizado por tentativa e erro se desenvolveu inicialmente na psicologia, mais especificamente por pesquisadores que estudavam como os animais aprendiam. Vale citar Edward Thorndike com sua “Law of Effect” e Pavlov, com o famoso cachorro de Pavlov. A partir desse momento, no final do século XIV e início do século XX, surgiu o conceito de “reforço”, e da perspectiva de aprendizagem a partir de recompensas e punições. Um agente tenderia a repetir um comportamento que é recompensado e inibir um comportamento que é punido.

Esses conceitos influenciaram a área de inteligência artificial desde Alan Turing, que imaginava um design de “prazer e dor” para orientar o comportamento de um sistema. Alguns outros pesquisadores chegaram a desenvolver sistemas que fossem capazes de aprender por tentativa e erro, como Thomas Ross e Grey Walter. Um trabalho que vale a pena ser citado foi o de **Claude Shannon** que desenvolveu um sistema magnético capaz de resolver um labirinto. Vale a pena citar também o SNARCs, feito por **Marvin Minsky**, que pode ser considerado um protótipo de máquina que aprende por reforço.

A partir daí, no final dos anos 50, a área de Aprendizado por Reforço entrou em um inverno que durou algumas décadas, onde pouco avanço foi feito no sentido de aprender por tentativa e erro. Algumas poucas contribuições relevantes aconteceram e cabe citar aqui o algoritmo MENACE de **Donald Michie** que aprendeu a jogar o jogo da velha utilizando uma abstração de feijões coloridos em uma caixinha, onde cada cor representava um possível estado do jogo e a atualização da quantidade feijões de determinada cor em uma caixinha era atualizada de acordo com o resultado final do jogo. Essa noção foi expandida e aplicada para um pêndulo invertido.

Alguns avanços também foram feitos em campos como Learning Automatas, Aprendizado por Reforço aplicado à Economia e Teoria dos Jogos. Porém esses foram avanços muito interligados com suas respectivas áreas de aplicação

Controle Ótimo

A segunda linha do desenvolvimento de Aprendizado por Reforço remonta a uma área chamada Controle Ótimo. **Richard Bellman** fez desenvolvimentos importantes nessa área que mais tarde seriam usados como fundamentos para a modelagem matemática do Aprendizado por Reforço mais tarde. Dentre essas contribuições podemos citar a Programação Dinâmica, que é um método matemático para resolução de problemas de otimização, a noção de Função de Valor, que serve para determinar a qualidade do estado atual do problema e os Processos de Decisão de Markov, que conseguem abstrair matematicamente um problema de decisão sequencial onde o próximo estado é determinado pela ação tomada no estado atual.

Essas contribuições importantes não foram conectadas com a noção de aprender por tentativa e erro até o final dos anos 80, quando uma outra linha de desenvolvimento do Aprendizado por reforço entrou em alta e uniu todas as linhas em uma só.

Temporal Difference

A última linha de Aprendizado por Reforço, conhecida como diferença temporal, se baseava na ideia de aprender durante uma experiência. Um trabalho importante demonstrando esse conceito foi o de Arthur Samuel em 1959 com seu programa que jogava damas a partir de uma Função de Valor que era atualizada durante a partida e não apenas no final. Essa área também passou por um inverno até que Ian Witten em 1977 e Barto e Sutton no final dos anos 1970 e começo dos anos 1980 fizeram algumas contribuições relevantes. Em 1989 **Chris Watkin** publicou seu algoritmo que juntava todos os três conceitos de aprender por tentativa e erro, resolver problemas de otimização e diferença temporal em um algoritmo conhecido como Q-Learning. A partir daí começava o Aprendizado por Reforço como conhecemos hoje.

NOTEBOOK: FUNÇÃO DE VALOR

```
!pip install gymnasium==0.29.1
```

```
Collecting gymnasium==0.29.1
  Downloading gymnasium-0.29.1-py3-none-any.whl.metadata (10 kB)
Requirement already satisfied: numpy>=1.21.0 in
/usr/local/lib/python3.10/dist-packages (from gymnasium==0.29.1) (1.26.4)
Requirement already satisfied: cloudpickle>=1.2.0 in
/usr/local/lib/python3.10/dist-packages (from gymnasium==0.29.1) (2.2.1)
Requirement already satisfied: typing-extensions>=4.3.0 in
/usr/local/lib/python3.10/dist-packages (from gymnasium==0.29.1) (4.12.2)
Collecting farama-notifications>=0.0.1 (from gymnasium==0.29.1)
  Downloading Farama_Notifications-0.0.4-py3-none-any.whl.metadata (558
bytes)
Downloading gymnasium-0.29.1-py3-none-any.whl (953 kB)
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 953.9/953.9 kB 13.9 MB/s eta
0:00:00
a_Notifications-0.0.4-py3-none-any.whl (2.5 kB)
Installing collected packages: farama-notifications, gymnasium
Successfully installed farama-notifications-0.0.4 gymnasium-0.29.1
```

```
import numpy as np
import warnings

np.set_printoptions(linewidth=np.inf)
warnings.filterwarnings("ignore", category=DeprecationWarning)
```

Abaixo temos um ambiente básico para um Processo de Decisão de Markov utilizando 5 estados e 2 ações possíveis.

```
import gymnasium as gym
from gymnasium import spaces

N = 5
A = 2
class MarkovDecisionProcess(gym.Env):
    def __init__(self):
        self.N = N
        self.states = [s for s in range(N)]
        self.transition_matrix = [
            [
                [0.00, 0.25, 0.25, 0.25, 0.25],
                [0.00, 0.00, 1/3, 1/3, 1/3],
                [0.00, 0.00, 0.00, 0.50, 0.50],
                [0.00, 0.00, 0.00, 0.00, 1.00],
```

```
                [0.00, 0.00, 0.00, 0.00, 0.00]],
            [
                [0.00, 1.00, 0.00, 0.00, 0.00],
                [0.00, 0.00, 1.00, 0.00, 0.00],
                [0.00, 0.00, 0.00, 1.00, 0.00],
                [0.00, 0.00, 0.00, 0.00, 1.00],
                [0.00, 0.00, 0.00, 0.00, 0.00]],
            ]
        self.reward_function = [0.0, 0.0, 0.0, 0.0, 1.0]

        self.observation_space = spaces.Discrete(N)
        self.action_space = spaces.Discrete(A)

        self.starting_state = 0
        self.current_state = 0
        self.timestep = 0
        self.terminal_states = [4]

    def step(self, action):

        next_state = np.random.choice(self.states,
p=self.transition_matrix[action][self.current_state])
        obs = next_state
        reward = self.reward_function[next_state]
        done = (next_state in self.terminal_states)
        self.timestep += 1
        info = {}
        self.current_state = next_state
        trunc = False

        return obs, reward, done, trunc, info

    def reset(self, seed=None, options=None):
        self.timestep = 0
        self.current_state = self.starting_state
        obs = self.starting_state
        info = {}
        return obs, info
```

Ambientes no OpenAI Gym são facilmente criados por um nome, e ambientes customizados podem ser registrados para compartilhar desta facilidade.

```
gym.envs.register(
     id='MDP-v0',
```

```
    entry_point="__main__:MarkovDecisionProcess",
)
```

A biblioteca Gym nos permite consultar alguns atributos dos ambientes, em especial os espaços de ação e observação:

```
env = gym.make("MDP-v0")
print(env.action_space)
print(env.observation_space)
```

```
Discrete(2)
Discrete(5)
```

Espaços podem assumir diversas formas, os mais comuns são:

- **Box**: para espaços multidimensionais e contínuos (imagens caem nesta categoria)
- **Discrete**: para espaços discretos
- **Dict**: para espaços customizados (geralmente multiagentes)
- **Text**: para espaços textuais nao numéricos

Fluxo de Interação

image.png

O fluxo de interação entre um agente e um ambiente durante um episódio é codificado da seguinte forma:

```
# Ciclo de Interação Agente-Ambiente
def run_episode():
    states = []
    actions = []
    rewards = []

    s, _ = env.reset()
    states.append(s)
    rewards.append(0.0)
    done = False
    while not done:

        a = env.action_space.sample() # Escolhendo ação aleatória
        next_s, r, done, trunc, info = env.step(a)
        done = done or trunc

        states.append(next_s)
```

```
        actions.append(a)
        rewards.append(r)

        s = next_s

    env.close() # Chamado somente quando o ambiente não será mais usado
(será necessário outro env.make)
    return states, actions, rewards
```

Cálculo do Retorno

Questão 1

Faça o cálculo do retorno e execute-o na trajetória adquirida acima com `GAMMA = 0.9`

image.png

```
# Cálculo do Retorno

states, actions, rewards = run_episode()

GAMMA = 0.9

def R(rewards, gamma):
    if len(rewards) > 0:
        return rewards[0] + gamma * R(rewards[1:], gamma)
    else:
        return 0

def returns_none(rewards, states, total_states, gamma):
    r_list = []
    ç = 0
    for i in range(total_states):
        if i in states:
            r_list.append(R(rewards[ç:], gamma))
            ç += 1
        else:
            r_list.append(None)
    return r_list

#returns([R(rewards[i:], 0.9) for i in range(len(rewards))], 5)
R(rewards, GAMMA)

returns_none(rewards, states, 5, GAMMA)
```

```
[0.7290000000000001, 0.81, 0.9, None, 1.0]
```

```
print(states)
print(actions)
print(rewards)
```

```
[0, 1, 2, 4]
[1, 1, 0]
[0.0, 0.0, 0.0, 1.0]
```

```
print(states)
```

```
[0, 1, 2, 4]
```

```
# Cálculo do Retorno
def retorno(states, rewards, gamma=0.9, total_states=5):
  R_list = []
  for x in range(len(rewards)):
    R = 0
    for i, r in enumerate(rewards[x:]):
      R += r*(gamma**i)
    R_list.append(R)

  while len(R_list) < total_states:
    for y in range(len(rewards)):
      if y not in states:
        R_list.insert(y, None)

  return R_list
```

Funções de Valor

Dado o ambiente demonstrado acima, abaixo estão as funções de valor para a política ótima e aleatória

image.png

```
# Função de valor ótima

V_star = [0.806, 0.827, 0.855, 0.9, 1.0]
Qsa_star = [[0.806, 0.743],[0.827, 0.769],[0.855, 0.81],[0.9, 0.9],[None,
None]]

#Função de Valor para Política Aleatória

V_random = [0.742, 0.784, 0.832, 0.9, 1.0]
Qsa_random = [[0.791, 0.705],[0.819, 0.749],[0.855, 0.81],[0.9, 0.9],[None,
None]]
```

Fluxo para múltiplos episódios

Questão 2

Utilizando o fluxo de interação agente-ambiente e o cálculo de retorno, atualize a célula a seguir para que ela execute vários episódios fazendo o cálculo do retorno para cada um deles.

Utilizando os retornos, calcule funções de valor (lista e matriz) para estados e pares estado-ação que sejam a média dos retornos de todos os episódios.

Execute o algoritmo mostrando no final o valor convergido pelas médias e compare com as funções de valor ótima e de política aleatória.

```
# Interação de vários episódios
env = gym.make("MDP-v0")

N_EPISODES = 10000
GAMMA = 0.9

MEDIA_RETORNOS = [0, 0, 0, 0, 0]
MEDIA_RETORNOS_COUNT = [0, 0, 0, 0, 0]

Q_sa = [[0, 0], [0, 0], [0, 0], [0, 0]]
Q_sa_count = [[0, 0], [0, 0], [0, 0], [0, 0]]

for ep in range(N_EPISODES):

# Ciclo de Interação Agente-Ambiente
    states = []
    actions = []
    rewards = []

    s = env.reset()[0]
    states.append(s)
    rewards.append(0)
    done = False
    while not done:

        a = env.action_space.sample() # Escolhendo ação aleatória
        next_s, r, done, trunc, info = env.step(a)
        done = done or trunc

        states.append(next_s)
        actions.append(a)
```

```
        rewards.append(r)

        s = next_s

    r_list = returns_none(rewards, states, 5, GAMMA)
    #print(r_list)
    for i, r in enumerate(r_list):

        #print(f'{i} - {r}')
        if r is not None:
            MEDIA_RETORNOS[i] += r
            MEDIA_RETORNOS_COUNT[i] += 1
            if i < 4:
                j = actions.pop(0)
                Q_sa[i][j] += r
                Q_sa_count[i][j] += 1

env.close()

MEDIA_RETORNO_TOTAL = [mr/mrc for mr, mrc in zip(MEDIA_RETORNOS,
MEDIA_RETORNOS_COUNT)]
Q_sa_total = [[qs[0]/qsc[0], qs[1]/qsc[1]] for qs, qsc in zip(Q_sa,
Q_sa_count)]
print(MEDIA_RETORNO_TOTAL)
print(Q_sa_total)
```

```
[0.7473540600000791, 0.7835144490314387, 0.8324345930232592,
0.8999999999999412, 1.0]
[[0.7899086486486351, 0.7048844955044923], [0.8174896095717811,
0.7489516335682179], [0.8557842046718597, 0.8099999999999582],
[0.9000000000000521, 0.9000000000000534]]
```

Questão 3

Agora, faça uma simples função onde, dado um estado e a função de valor estado-ação, ela retorna a ação que maximize o Q(s,a). Porém, 5% das vezes em que é chamada, ela escolhe uma ação aleatória.

```
import random

def pick_a(s, Q, epsilon):
    if random.random() < epsilon:
        value = random.choice(Q[s])
        return Q[s].index(value)
    else:
```

```
        value = max(Q[s])
        return Q[s].index(value)
```

Questão 4

Copie o código de múltiplos episódios executado anteriormente, porém, faça com que a ação seja escolhida pela função acima.

Obs: Lembre-se de atualizar Q(s,a) com as médias dos retornos a cada episódio.

```
# Interação de vários episódios
env = gym.make("MDP-v0")


def run_it(N_EPISODES, GAMMA, EPSILON):
    V = [0, 0, 0, 0, 0]
    V_count = [0, 0, 0, 0, 0]

    Q = [[0, 0], [0, 0], [0, 0], [0, 0]]
    Q_count = [[0, 0], [0, 0], [0, 0], [0, 0]]

    for ep in range(N_EPISODES):

    # Ciclo de Interação Agente-Ambiente
        states = []
        actions = []
        rewards = []

        s = env.reset()[0]
        states.append(s)
        rewards.append(0)
        done = False
        while not done:

            a = pick_a(s, Q, EPSILON)
            next_s, r, done, trunc, info = env.step(a)
            done = done or trunc

            states.append(next_s)
            actions.append(a)
            rewards.append(r)

            s = next_s

        r_list = returns_none(rewards, states, 5, GAMMA)
        #print(r_list)
```

```
        for i, r in enumerate(r_list):

            #print(f'{i} - {r}')
            if r is not None:
                V[i] += r
                V_count[i] += 1
                if i < 4:
                    j = actions.pop(0)
                    Q[i][j] += r
                    Q_count[i][j] += 1

    env.close()
    V_mean = [mr/mrc for mr, mrc in zip(V, V_count)]
    Q_mean = [[qs[0]/qsc[0], qs[1]/qsc[1]] for qs, qsc in zip(Q, Q_count)]

    return V_mean, Q_mean

V, Q = run_it(10000, 0.9, 0.05)
```

```
V
```

```
[0.8033194800000694,
 0.8232383720930185,
 0.8533333333333357,
 0.9000000000000239,
 1.0]
```

```
Q
```

```
[[0.804811293129994, 0.7407849785407746],
 [0.8247398130841082, 0.7710779220779224],
 [0.8545476698141977, 0.810000000000001],
 [0.900000000000035, 0.9000000000000022]]
```

Questão 5

Altere o valor de gamma para calcular o retorno e veja como isso altera as funções de valor. Coloque abaixo os resultados adquiridos indicando gamma e os valores.

```
import numpy as np
import plotly.graph_objs as go

# Step 1: Define the function that returns 5 values
def my_function(gamma):
    return [np.sin(gamma + i) for i in range(5)]

# Step 2: Create a range of gamma values
```

```
gamma_values = np.linspace(0.01, 1, 100)  # From 0 to 10 with 100 steps

# Step 3: Compute the function for each gamma value
function_values = [run_it(10000, gamma, 0.05)[0] for gamma in gamma_values]

# Step 4: Prepare data for Plotly
traces = []
for i in range(5):  # Each list index corresponds to a separate trace
    trace = go.Scatter(
        x=gamma_values,
        y=[values[i] for values in function_values],
        mode='lines',
        name=f'Value {i+1}'
    )
    traces.append(trace)

# Step 5: Create the layout and figure
layout = go.Layout(
    title='Evolution of Function Based on Gamma',
    xaxis=dict(title='Gamma'),
    yaxis=dict(title='Function Values')
)

fig = go.Figure(data=traces, layout=layout)

# Step 6: Show the plot
fig.show()
```

Modelagem

Questão 6

Imagine a seguinte situação:

Um navio pesqueiro automatizado decide suas ações baseando-se na quantidade de peixes detectados na região por imagens de satélite. A quantidade de peixes varia entre **nenhum**, **baixa**, **media**, e **alta**. Ao ler estas informações, o computador do navio precisa decidir se irá **pescar** ou **não**.

Lembrando que **pescar** custa combustível (ou seja, dinheiro), portanto não é prudente **pescar** a todo momento. Ao **pescar** com qualquer nível de peixes, existem 3/4 de chance da população diminuir para um nível anterior e 1/4 de permanecer no mesmo nível. **Não pescar** também possui seu custo com pessoal, mas são menores que os de **pescar** sem pegar nada. Ao **não pescar**, os peixes são permitidos de reproduzir e aumentar sua população sendo que de **nenhum** para **baixa** é 100% de chance, de **baixa** para **média** há 75% de chance, e de **média** para **alta** há 60% de chance. Ao

pescar com o nível **baixo**, os lucros da pesca passam levemente dos custos de operação do navio. Ao **pescar** com o nível **médio** os lucros cobrem o custo de operação do barco duas vezes, ao **pescar** em nível **alto**, os lucros são dobrados em relação aos de pescar em nível médio.

O navio opera por **12 meses** tomando decisões de pesca a cada mês. O objetivo é, portanto, maximizar os lucros ao longo do ano. Vale lembrar que, em janeiro, devido ao clima frio e migração, nunca há peixes detectados pelo satélite.

Faça o código do ambiente descrito acima na célula abaixo.

Obs: Lembre-se de comentar as alterações no código do ambiente para facilitar a leitura.

```
N = 4 # numero de estados
A = 2 # numero de ações
class MarkovDecisionProcessv1(gym.Env):
    def __init__(self):
        self.N = N
        self.states = [s for s in range(N)]
        self.transition_matrix = [
            [
                [0.00, 1.00, 0.00, 0.00],
                [0.00, 0.25, 0.75, 0.00],
                [0.00, 0.00, 0.40, 0.60],
                [0.00, 0.00, 0.00, 1.00]],
            [
                [1.00, 0.00, 0.00, 0.00],
                [0.75, 0.25, 0.00, 0.00],
                [0.00, 0.75, 0.25, 0.00],
                [0.00, 0.00, 0.75, 0.25]],
            ]
        self.reward_function = [-.25, .35, .5, 1.]

        self.observation_space = spaces.Discrete(N)
        self.action_space = spaces.Discrete(A)
        self.starting_state = 0
        self.current_state = 0
        self.timestep = 0

    def step(self, action):

        next_state = np.random.choice(self.states,
p=self.transition_matrix[action][self.current_state])
        obs = next_state
        reward = self.reward_function[next_state]
```

```
        self.timestep += 1
        if self.timestep > 10:
            done = True
        else:
            done = False
        info = {}
        self.current_state = next_state
        trunc = False

        return obs, reward, done, trunc, info

    def reset(self, seed=None, options=None):

        self.timestep = 0
        self.current_state = self.starting_state
        obs = self.starting_state
        info = {}
        return obs, info

gym.envs.register(
     id='MDP-v1',
     entry_point="__main__:MarkovDecisionProcessv1",
)
```

```
/usr/local/lib/python3.10/dist-packages/gymnasium/envs/registration.py:694:
UserWarning:

WARN: Overriding environment MDP-v1 already in registry.
```

Questão 7

Utilize o código da Questão 5 para que a política atue no ambiente da Questão 6 e observe os resultados.

Obs: Lembre-se de observar os valores convergidos para averiguar se eles fazem sentido com as dinâmicas do problema.

```
# Interação de vários episódios
env = gym.make("MDP-v1")


states = []
actions = []
rewards = []
```

```
s, _ = env.reset()
states.append(s)
rewards.append(0.0)
done = False
while not done:

    a = env.action_space.sample() # Escolhendo ação aleatória
    next_s, r, done, trunc, info = env.step(a)
    done = done or trunc

    states.append(next_s)
    actions.append(a)
    rewards.append(r)
    s = next_s

env.close() # Chamado somente quando o ambiente não será mais usado (será necessário outro env.make)
print(states)
print(actions)
print(rewards)
```

```
[0, 1, 0, 0, 1, 1, 2, 1, 2, 1, 0, 1]
[0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0]
[0.0, 0.0, -1.0, -1.0, 0.0, 0.0, 1.0, 0.0, 1.0, 0.0, -1.0, 0.0]
```

```
list1 = ['a', 'b', 'c']
list2 = [1, 2, 3]

for index, (item1, item2) in enumerate(zip(list1, list2)):
    print(f"Index: {index}, List1 item: {item1}, List2 item: {item2}")
```

```
Index: 0, List1 item: a, List2 item: 1
Index: 1, List1 item: b, List2 item: 2
Index: 2, List1 item: c, List2 item: 3
```

```
[[1,2]][0][0]
```

```
1
```

```
# Cálculo do Retorno

states, actions, rewards = run_episode()

GAMMA = 0.9

def R(rewards, gamma):
    total = 0
```

```
    for i, reward in enumerate(rewards):
        total += reward * (gamma ** i)
    return total

def VQ_(states, actions, rewards, n_states, n_actions, gamma):
    V = [0] * n_states
    V_count = [0] * n_states
    Q = [[0 for _ in range(n_actions)] for _ in range(n_states)]
    Q_count = [[0 for _ in range(n_actions)] for _ in range(n_states)]

    for index, (state, action) in enumerate(zip(states[:-1], actions)):
        V[state] += R(rewards[index:], gamma)
        V_count[state] += 1
        Q[state][action] += R(rewards[index:], gamma)
        Q_count[state][action] += 1

    V[states[-1]] += R([rewards[-1]], gamma)
    V_count[states[-1]] += 1

    V = [v / max(c, 1) for v, c in zip(V, V_count)]
    Q = [[q / max(c, 1) for q, c in zip(qs, counts)] for qs, counts in
zip(Q, Q_count)]

    return V, Q

#returns([R(rewards[i:], 0.9) for i in range(len(rewards))], 5)
R(rewards, GAMMA)

V, Q = VQ_(states, actions, rewards, 4, 2, GAMMA)
print("V:", V)
print("Q:", Q)
```

```
V: [2.6100381737175007, 3.7414209990000002, 4.32777901, 4.35]
Q: [[3.0105664986500003, 2.2095098487850007], [4.4561849985,
2.3118930000000004], [4.951316665, 3.0807037], [5.7, 0.0]]
```

```
def run_it(N_EPISODES, GAMMA, EPSILON, n_states, n_actions):

    V = [0] * n_states
    V_count = [0] * n_states
    Q = [[0 for _ in range(n_actions)] for _ in range(n_states)]
    Q_count = [[0 for _ in range(n_actions)] for _ in range(n_states)]

    for ep in range(N_EPISODES):
```

```
    # Ciclo de Interação Agente-Ambiente
        states = []
        actions = []
        rewards = []

        s = env.reset()[0]
        states.append(s)
        rewards.append(0)
        done = False
        while not done:

            a = pick_a(s, Q, EPSILON)
            next_s, r, done, trunc, info = env.step(a)
            done = done or trunc

            states.append(next_s)
            actions.append(a)
            rewards.append(r)

            s = next_s

        V_, Q_ = VQ_(states, actions, rewards, n_states, n_actions, GAMMA)

        for i, v in enumerate(V_):
            V[i] += v
            V_count[i] += 1

        for i, q_values in enumerate(Q_):
            for j, q in enumerate(q_values):
                Q[i][j] += q
                Q_count[i][j] += 1


    env.close()

    V_mean = [v / max(count, 1) for v, count in zip(V, V_count)]
    Q_mean = [[q / max(count, 1) for q, count in zip(q_row, count_row)] for
q_row, count_row in zip(Q, Q_count)]

    return V_mean, Q_mean

V, Q = run_it(10000, 0.9, 0.05, 4, 2)
print("V:", V)
print("Q:", Q)
```

```
V: [12.099850823566477, 13.716922899332605, 15.171917259339956,
10.43060362864143]
Q: [[12.1158087976558, 0.264276452754198], [13.755984790220769,
0.3079120872622052], [15.321124524504514, 0.4958344433286511],
[11.540115542037249, 1.5793296848189877]]
```

```python
import numpy as np
import plotly.graph_objs as go

# Step 1: Define the function that returns 5 values
def my_function(gamma):
    return [np.sin(gamma + i) for i in range(5)]

# Step 2: Create a range of gamma values
gamma_values = np.linspace(0.01, 1, 100)  # From 0 to 10 with 100 steps

# Step 3: Compute the function for each gamma value
function_values = [run_it(10000, gamma, 0.05, 4, 2)[0] for gamma in
gamma_values]

# Step 4: Prepare data for Plotly
traces = []
for i in range(4):  # Each list index corresponds to a separate trace
    trace = go.Scatter(
        x=gamma_values,
        y=[values[i] for values in function_values],
        mode='lines',
        name=f'Value {i+1}'
    )
    traces.append(trace)

# Step 5: Create the layout and figure
layout = go.Layout(
    title='Evolution of Function Based on Gamma',
    xaxis=dict(title='Gamma'),
    yaxis=dict(title='Function Values')
)

fig = go.Figure(data=traces, layout=layout)

# Step 6: Show the plot
fig.show()
```

APÊNDICE 2

Termo de Aceite de Entrega

Objetivo deste documento

Este documento faz parte do Processo da disciplina Residência em IA e tem como objetivo formalizar o aceite da entrega considerando o planejado e o realizado para o período.

Data da Reunião ("gate") de aprovação: 26 de set. de 2024

Participantes da Entrega [matriculados em Residência em IA]:

ANDRÉ LUIS ARAÚJO DE SOUZA

Entrega: [descrever a ENTREGA: requisitos e produtos gerados: links para textos, códigos, vídeos etc.]

As atividades realizadas foram:

- Pesquisa de Artigos relevantes dentro da área de Aprendizado por Reforço;
- Compilação dos Artigos na seguinte lista: Lista de Artigos de Aprendizado por Reforço ;
- Elaboração de um pequeno texto contendo os conceitos fundamentais de Aprendizado por Reforço: Fundamentos do RL ;
- Hands-On nos frameworks mais populares:
  - Gymnasium: Gymnasium.ipynb ;
  - Stable Baselines: Stable Baselines.ipynb ;
  - Ray RLlib: Ray RLlib.ipynb ;
- Mapa mental representando o "landscape" do Aprendizado por Reforço: RL MindMap Sketch.png .

Planejamento: [descrever o que pretende fazer para realizar a próxima ENTREGA]

O planejamento para a próxima semana é:

- Seleção de Leitura de artigos relevantes na lista já compilada;
- Apresentar e discutir ideias no grupo de estudo de Aprendizado por Reforço da UFG;
- Realizar alterações incrementais na documentação produzida até aqui.

Observação: [caso precise fazer alguma observação, de qualquer "natureza"]

ACEITE DA ENTREGA:

CEDRIC LUIZ DE CARVALHO: Go!

NOTEBOOK: HANDS-ON FRAMEWORK GYMNASIUM

```
!pip install swig
!pip install gymnasium[box2d]
```

```
Requirement already satisfied: swig in
/usr/local/lib/python3.10/dist-packages (4.2.1)
Requirement already satisfied: gymnasium[box2d] in
/usr/local/lib/python3.10/dist-packages (0.29.1)
Requirement already satisfied: numpy>=1.21.0 in
/usr/local/lib/python3.10/dist-packages (from gymnasium[box2d]) (1.26.4)
Requirement already satisfied: cloudpickle>=1.2.0 in
/usr/local/lib/python3.10/dist-packages (from gymnasium[box2d]) (2.2.1)
Requirement already satisfied: typing-extensions>=4.3.0 in
/usr/local/lib/python3.10/dist-packages (from gymnasium[box2d]) (4.12.2)
Requirement already satisfied: farama-notifications>=0.0.1 in
/usr/local/lib/python3.10/dist-packages (from gymnasium[box2d]) (0.0.4)
Collecting box2d-py==2.3.5 (from gymnasium[box2d])
  Using cached box2d-py-2.3.5.tar.gz (374 kB)
  Preparing metadata (setup.py) ... ent already satisfied: pygame>=2.1.3 in
/usr/local/lib/python3.10/dist-packages (from gymnasium[box2d]) (2.6.0)
Requirement already satisfied: swig==4.* in
/usr/local/lib/python3.10/dist-packages (from gymnasium[box2d]) (4.2.1)
Building wheels for collected packages: box2d-py
  Building wheel for box2d-py (setup.py) ...
e=box2d_py-2.3.5-cp310-cp310-linux_x86_64.whl size=2376100
sha256=fce0bdf166b584827de96c3a57631a03c89509ad556601ff5f131dbea653f53b
  Stored in directory:
/root/.cache/pip/wheels/db/8f/6a/eaaadf056fba10a98d986f6dce954e6201ba312692
6fc5ad9e
Successfully built box2d-py
Installing collected packages: box2d-py
Successfully installed box2d-py-2.3.5
```

```
import gymnasium as gym
from gymnasium.wrappers import RecordVideo

env = RecordVideo(gym.make('LunarLander-v2', render_mode="rgb_array"),
video_folder="./video")

# Reset the environment and run for a few steps
obs, info = env.reset()
```

```
for _ in range(100):
    action = env.action_space.sample()  # Random action
    obs, reward, done, truncated, info = env.step(action)

    if done or truncated:
        obs, info = env.reset()

env.close()

# Display the video
import glob
from IPython.display import HTML
from base64 import b64encode

def show_video():
    mp4list = glob.glob('./video/*.mp4')
    if len(mp4list) > 0:
        mp4 = mp4list[1]
        video = open(mp4, "rb").read()
        data_url = "data:video/mp4;base64," + b64encode(video).decode()
        return HTML(f'<video width="400" controls><source src="{data_url}"
type="video/mp4"></video>')

show_video()
```

```
Moviepy - Building video /content/video/rl-video-episode-0.mp4.
Moviepy - Writing video /content/video/rl-video-episode-0.mp4



Moviepy - Done !
Moviepy - video ready /content/video/rl-video-episode-0.mp4
```

```
<IPython.core.display.HTML object>
```

NOTEBOOK: HANDS-ON FRAMEWORK STABLE BASELINES

```
!pip -q install swig
!pip -q install stable-baselines3[extra] gymnasium[box2d]
```

```
  Preparing metadata (setup.py) ...
```

```
import gymnasium as gym
from google.colab.patches import cv2_imshow
from stable_baselines3 import PPO
from stable_baselines3.common.env_util import make_vec_env

# Parallel environments
vec_env = make_vec_env("CartPole-v1", n_envs=4)

model = PPO("MlpPolicy", vec_env, verbose=1)
model.learn(total_timesteps=25000)
model.save("ppo_cartpole")

del model # remove to demonstrate saving and loading

model = PPO.load("ppo_cartpole")

obs = vec_env.reset()
while True:
    action, _states = model.predict(obs)
    obs, rewards, dones, info = vec_env.step(action)
    vec_env.render("rgb_array")
```

```
Saving video to
/content/logs/videos/random-agent-CartPole-v1-step-0-to-step-100.mp4
Moviepy - Building video
/content/logs/videos/random-agent-CartPole-v1-step-0-to-step-100.mp4.
Moviepy - Writing video
/content/logs/videos/random-agent-CartPole-v1-step-0-to-step-100.mp4
```

```
Moviepy - Done !
Moviepy - video ready
/content/logs/videos/random-agent-CartPole-v1-step-0-to-step-100.mp4
```

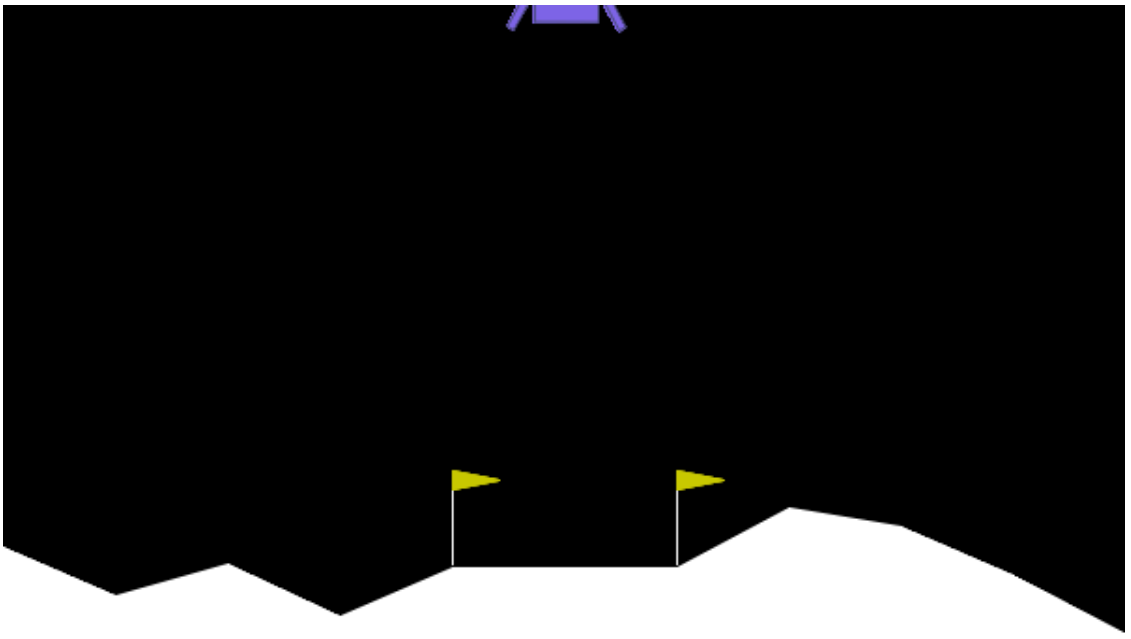
```
import gymnasium as gym
from stable_baselines3.common.vec_env import VecVideoRecorder, DummyVecEnv

env_id = "CartPole-v1"
video_folder = "logs/videos/"
video_length = 100

vec_env = DummyVecEnv([lambda: gym.make(env_id, render_mode="rgb_array")])

obs = vec_env.reset()

# Record the video starting at the first step
vec_env = VecVideoRecorder(vec_env, video_folder,
                           record_video_trigger=lambda x: x == 0,
video_length=video_length,
                           name_prefix=f"random-agent-{env_id}")

vec_env.reset()
for _ in range(video_length + 1):
  action = [vec_env.action_space.sample()]
  obs, _, _, _ = vec_env.step(action)
# Save the video
vec_env.close()
```

```
import imageio
import numpy as np

from stable_baselines3 import A2C

model = A2C("MlpPolicy", "LunarLander-v2").learn(100_000)

images = []
obs = model.env.reset()
img = model.env.render(mode="rgb_array")
for i in range(350):
    images.append(img)
    action, _ = model.predict(obs)
    obs, _, _ ,_ = model.env.step(action)
    img = model.env.render(mode="rgb_array")

imageio.mimsave("lander_a2c.gif", [np.array(img) for i, img in
enumerate(images) if i%2 == 0], fps=29)
```

```
/usr/local/lib/python3.10/dist-packages/imageio/plugins/pillow.py:409:
DeprecationWarning: The keyword `fps` is no longer supported. Use
```

```
`duration`(in ms) instead, e.g. `fps=50` == `duration=20` (1000 * 1/50).
  warnings.warn(
```

```
from IPython.display import Image

# Display the GIF
Image(filename='/content/lander_a2c.gif')
```



NOTEBOOK: HANDS-ON FRAMEWORK RAY RLLIB

```
!pip install -q "ray[rllib]" tensorflow torch
```

```
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0.0/925.5 kB ? eta
-:--:--━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 921.6/925.5 kB 30.5 MB/s
eta 0:00:01━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 925.5/925.5 kB 16.8
MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 101.7/101.7 kB 6.9 MB/s eta
0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.3/1.3 MB 41.8 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 65.6/65.6 MB 11.7 MB/s eta 0:00:00
```

```python
import ray
from ray import tune
from ray.rllib.algorithms.ppo import PPOConfig
import gymnasium as gym

# Step 1: Initialize Ray
ray.init(ignore_reinit_error=True)

# Step 2: Define the PPO Configuration
config = (
    PPOConfig()
    .environment(env="CartPole-v1")  # Set the Gymnasium environment
    .rollouts(num_rollout_workers=1) # Number of workers for collecting
data
    .framework('torch')              # Choose between "torch" or "tf" for
PyTorch or TensorFlow
    .training(train_batch_size=4000) # Number of steps to train on per
iteration
)

# Step 3: Build and Train the PPO Algorithm
algo = config.build()

# Step 4: Train the model for a few iterations
for i in range(10):
    result = algo.train()  # Train for one iteration
    print(f"Iteration {i+1}:
reward={result['env_runners']['episode_reward_mean']}")

# Step 5: Save the trained model
```

```
checkpoint_path = algo.save("./ppo_cartpole_checkpoint")
print(f"Checkpoint saved at {checkpoint_path}")
```

```
2024-09-26 14:08:56,264INFO worker.py:1619 -- Calling ray.init() again
after it has already been called.
2024-09-26 14:08:56,267WARNING deprecation.py:50 -- DeprecationWarning:
`AlgorithmConfig.env_runners(num_rollout_workers)` has been deprecated. Use
`AlgorithmConfig.env_runners(num_env_runners)` instead. This will raise an
error in the future!
2024-09-26 14:09:03,195WARNING util.py:61 -- Install gputil for GPU system
monitoring.
```

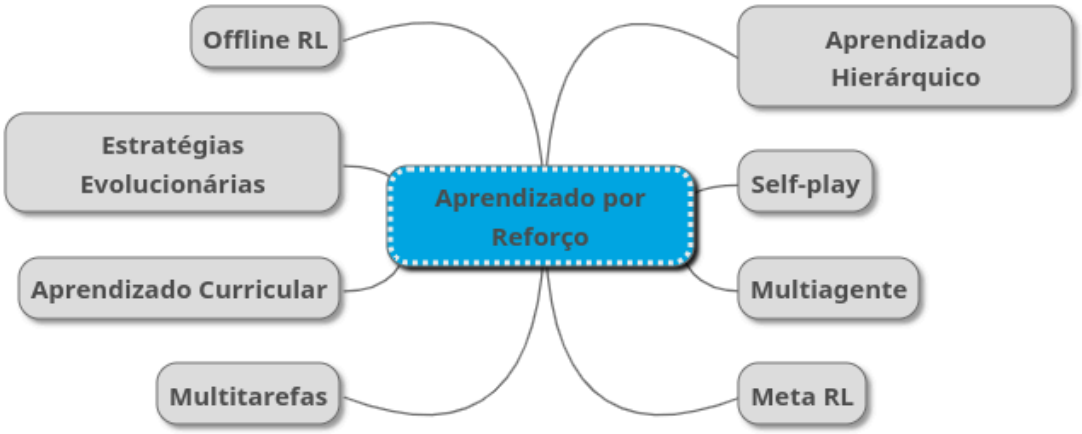
```
Iteration 1: reward=22.636363636363637
Iteration 2: reward=41.95
Iteration 3: reward=65.52
Iteration 4: reward=93.56
Iteration 5: reward=126.08
Iteration 6: reward=164.72
Iteration 7: reward=200.09
Iteration 8: reward=230.14
Iteration 9: reward=264.61
Iteration 10: reward=296.23
```

```
# Step 7: Restore the algorithm from checkpoint
algo.restore(checkpoint_path)

# Step 8: Test the trained policy
env = gym.make("CartPole-v1")
obs, info = env.reset()
done = False
total_reward = 0

while not done:
    action = algo.compute_single_action(obs)  # Use the trained policy
    obs, reward, done, truncated, info = env.step(action)
    total_reward += reward

print(f"Total reward after evaluation: {total_reward}")
```

```
2024-09-26 14:13:06,396INFO trainable.py:583 -- Restored on 172.28.0.12
from checkpoint: Checkpoint(filesystem=local,
path=./ppo_cartpole_checkpoint)
```

```
Total reward after evaluation: 21741.0
```

MAPA MENTAL: RAMIFICAÇÕES DO APRENDIZADO POR REFORÇO



Termo de Aceite de Entrega

Objetivo deste documento

Este documento faz parte do Processo da disciplina Residência em IA e tem como objetivo formalizar o aceite da entrega considerando o planejado e o realizado para o período.

Data da Reunião ("gate") de aprovação: 2 de out. de 2024

Participantes da Entrega [matriculados em Residência em IA]:

ANDRÉ LUIS ARAÚJO DE SOUZA

Entrega: [descrever a ENTREGA: requisitos e produtos gerados: links para textos, códigos, vídeos etc.]

Durante a Semana, foram realizadas as seguintes atividades:

- Leitura dos artigos:
  - Deep Reinforcement Learning: A Survey (Wang et al, 2022);
  - Deep Reinforcement Learning: A Brief Survey (Arulkumaran et al, 2017);
- Update no documento de fundamentos: Fundamentos do RL ;
- Update na lista de artigos com auxílio da ferramenta rabbit research: Lista de Artigos de Aprendizado por Reforço ;
- Discussão de ideias com grupo de estudo em RL do CEIA;
- Pesquisa de artigos sobre Aprendizado por Reforço Multiagente.

Planejamento: [descrever o que pretende fazer para realizar a próxima ENTREGA]

Os próximos passos planejados consistem em:

- Leitura do artigo Multi-Agent Reinforcement Learning: A Review of Challenges and Applications (Canese et al, 2021);
- Estudo de frameworks e bibliotecas em cenários multiagentes;

Observação: [caso precise fazer alguma observação, de qualquer "natureza"]

ACEITE DA ENTREGA:

CEDRIC LUIZ DE CARVALHO: Go!

Fundamentos do Aprendizado por Reforço

O objetivo desse documento é mapear os conceitos básicos da área de Aprendizado por Reforço. A ideia é que alguém que se considere um especialista nessa área precisa ter, necessariamente, o domínio desses conceitos.

O que é Aprendizado por Reforço?

Aprendizado por Reforço é um paradigma de aprendizagem de máquina, onde temos a presença de um agente que vai aprender através de experiências. Aprendizado por Reforço se contrapõe a outros dois paradigmas: Aprendizado Supervisionado, onde o aprendizado acontece a partir de um conjunto de dados rotulados, e Aprendizado Não-Supervisionado, onde o objetivo é aprender padrões ocultos em conjuntos de dados não anotados.

Princípios

Um problema de Aprendizado por Reforço tem como elementos estruturantes os seguintes:

- **Agente:** entidade que vai agir em um ambiente.
- **Ambiente:** espaço onde o agente vai agir.
- **Estado:** configuração atual do ambiente.
- **Ação:** decisão do agente em um determinado momento.
- **Recompensa:** sinal recebido pelo agente ao chegar em determinado estado.
- **Política:** função que mapeia estados para ações - define o comportamento do agente.
- **Função de Valor:** função que descreve a qualidade de um estado ou par estado-ação.
- **Modelo:** módulo que visa aprender a dinâmica do ambiente.
- **Episódio:** Uma amostra de experiência da interação de um agente com seu respectivo ambiente.
- **Trajetória:** Sequência de estados, ações e recompensas em um determinado episódio.

Formulação Matemática

- **Processo de Decisão de Markov:** os problemas de Aprendizado por Reforço são, geralmente, modelados como um Processo de Decisão de Markov - que engloba os conceitos principais enunciados anteriormente.
- **Probabilidade de Transição:** é a probabilidade que se tem de se chegar em um determinado estado a partir de uma ação.

- **Propriedade Markoviana:** em um problema de Aprendizado por Reforço, o passado não importa - todas as informações para se tomar uma ação precisa estar no estado atual.
- **Política:** pode seguir a função de valor ou ser um modelo a parte - seu objetivo é ser uma função de distribuição de probabilidade condicional, onde o agente amostra uma ação a partir de um determinado estado.
- **Retorno:** recompensa acumulada ao longo da trajetória - é mais informativo do que a recompensa pura - é definido a partir de um parâmetro GAMMA que pondera o quanto o futuro impacta no estado atual.
- **Função de Valor:** originalmente era enunciada como uma **Equação de Bellman** onde o valor do estado atual é a expectância estatística de retorno de um estado. Também pode ser um modelo aprendido.
- **Função de Recompensa:** função que mapeia uma transição de estado a um sinal de recompensa - é definida por quem está modelando o problema e vai ditar o processo de aprendizagem.

Terminologias

Algoritmos de Aprendizado por Reforço podem ser classificados como:

- **Model-Based and Model-Free:** Algoritmos Model-Based utilizam de um ***Modelo*** de mundo, enquanto os Model-Free não.
- **Policy Based and Value Based:** Algoritmos Policy-Based realizam o aprendizado tentando encontrar uma ***Política*** ótima, enquanto os Value-Based procuram uma ***Função de Valor*** ótima.
- **On-Policy and Off-Policy:** Algoritmos On-Policy são algoritmos que dependem exclusivamente da experiência da política atual para serem atualizados, enquanto os Off-Policy podem ser atualizados a partir da experiência de qualquer política.

Aprendizado por Reforço Profundo

Redes Neurais Artificiais são uma técnica de machine learning utilizada para aproximar funções. Ao combinar essa técnica com Aprendizado por Reforço, entramos no Aprendizado por Reforço Profundo (DRL). Aqui usamos Redes Neurais Profundas para aproximar funções complexas, como a Função de Valor, ou a própria Política.

De acordo com [Wang et al, 2022] podemos classificar o Aprendizado por Reforço Profundo em três categorias:

Value-based DRL

No Value-based DRL, o foco está em aprender uma função de valor, que estima o valor esperado de um estado ou de uma ação em um estado particular. A técnica mais conhecida nessa categoria é o Deep Q-Networks (DQN). Essa abordagem é eficiente em ambientes discretos, onde o número de ações é gerenciável, mas pode se tornar complexa para ações contínuas.

Policy-based DRL

No Policy-based DRL, o agente aprende diretamente uma política, que é uma função que mapeia estados para probabilidades de escolher cada ação. Ao invés de aprender uma função de valor e derivar a política, como no Value-based DRL, a política é parametrizada diretamente por uma rede neural e otimizada para maximizar a recompensa total esperada. Métodos como o REINFORCE e o Proximal Policy Optimization (PPO) são exemplos dessa abordagem. Policy-based DRL é vantajoso em cenários com ações contínuas ou onde é difícil derivar a política a partir da função de valor.

Maximum Entropy DRL

No Maximum Entropy DRL, como no método Soft Actor-Critic (SAC), a ideia é otimizar não apenas a recompensa esperada, mas também a entropia da política, promovendo a exploração. A entropia é uma medida de incerteza, e maximizar a entropia significa incentivar o agente a ser mais imprevisível e explorar uma variedade maior de ações. Esse enfoque ajuda o agente a evitar soluções subótimas, equilibrando exploração e exploração. Maximum Entropy DRL é especialmente útil em ambientes onde a exploração é crítica, levando a comportamentos mais robustos e com maior generalização.

Tópicos Avançados de Aprendizado por Reforço

Aprendizado por Reforço Baseado em Modelo

O Aprendizado por Reforço Baseado em Modelo utiliza uma representação interna do ambiente (modelo) para prever os resultados de ações e estados futuros. Com isso, o agente pode planejar ações e simular cenários antes de executá-las no ambiente real, acelerando o aprendizado.

Aprendizado por Reforço Hierárquico

O Aprendizado por Reforço Hierárquico divide tarefas complexas em sub-tarefas menores, onde políticas de baixo nível resolvem essas subtarefas, e políticas de alto nível coordenam as ações, permitindo ao agente resolver problemas mais complexos de forma estruturada.

Aprendizado por Reforço Multiagente

No Aprendizado por Reforço Multiagente, múltiplos agentes aprendem simultaneamente em um ambiente compartilhado. Cada agente interage com os outros e ajusta sua política para maximizar sua recompensa, considerando a cooperação ou competição entre eles.

Meta-aprendizado

Meta-aprendizado envolve treinar um modelo para aprender novas tarefas com menos dados ou interações. O objetivo é construir agentes que possam generalizar rapidamente para novos ambientes ou tarefas a partir de sua experiência anterior.

Aprendizado por Reforço Multitarefas

O Aprendizado por Reforço Multitarefas foca no desenvolvimento de agentes que podem aprender e executar múltiplas tarefas diferentes em um único ambiente. O objetivo é compartilhar conhecimento entre as tarefas para melhorar a eficiência do aprendizado em cada uma delas.

Aprendizado por Reforço Offline

O Aprendizado por Reforço Offline treina um agente com base em um conjunto fixo de dados coletados previamente, sem interagir diretamente com o ambiente durante o processo de treinamento. Isso é útil em cenários onde a coleta de novos dados é cara ou arriscada.

Self-play

No Self-play, o agente aprende jogando contra versões anteriores de si mesmo. Esse método é especialmente eficaz em jogos competitivos, onde o agente melhora progressivamente ao enfrentar adversários mais desafiadores ao longo do tempo.

Estratégias Evolucionárias

As Estratégias Evolucionárias são uma abordagem inspirada na evolução biológica para otimizar políticas em Aprendizado por Reforço. Elas utilizam mecanismos como mutação, recombinação e seleção para explorar o espaço de políticas, permitindo a descoberta de soluções eficientes sem a necessidade de gradientes.

APÊNDICE 3

Termo de Aceite de Entrega

Objetivo deste documento

Este documento faz parte do Processo da disciplina Residência em IA e tem como objetivo formalizar o aceite da entrega considerando o planejado e o realizado para o período.

Data da Reunião ("gate") de aprovação: 10 de out. de 2024

Participantes da Entrega [matriculados em Residência em IA]:

ANDRÉ LUIS ARAÚJO DE SOUZA

Entrega: [descrever a ENTREGA: requisitos e produtos gerados: links para textos, códigos, vídeos etc.]

Durante a Semana, foram realizadas as seguintes atividades:

- Leitura dos artigos:
  - A Comprehensive Survey of Multiagent Reinforcement Learning (Buşoniu et al, 2008);
  - Multi-Agent Reinforcement Learning: A Review of Challenges and Applications (Canese et al, 2021);
- Resumo dos artigos: Aprendizado por Reforço Multiagente ;
- Consulta a outros artigos:
  - Multi-agent deep reinforcement learning: a survey (Gronauer & Dielpold, 2021);
  - The Complexity of Decentralized Control of Markov Decision Processes (Bernstein et al, 2002);
- Pesquisa sobre frameworks para MARL:
  - https://pettingzoo.farama.org/;
  - https://magent2.farama.org/;
  - https://marllib.readthedocs.io/en/latest/.
- Leitura do Capítulo 2 do livro Reinforcement Learning: An Introduction.pdf .
- Esboço da definição de um ambiente para atacar:
  - N agentes com I, J, K unidades iniciais de $ABC, $XYZ e $OOO.
  - A cada X passos, os agentes pagam Q $OOO ou morrem.
  - A cada Y passos, os agentes recebem R $OOO, com rendimentos subsequentes de 100(alpha^n), onde 0 < alpha < 1.
  - Os agentes podem colocar e preencher ordens nos pares ABC/XYZ, ABC/OOO e XYZ/OOO.
  - Os agentes podem fechar suas próprias ordens.
  - Cada agente vê o preço dos ativos, o livro de ordens e seu próprio saldo.

Planejamento: [descrever o que pretende fazer para realizar a próxima ENTREGA]

Os próximos passos planejados consistem em:

- Realizar experimentos de treino de algoritmos em ambientes multiagentes utilizando os frameworks estudados;

- Pesquisar problemas multiagentes para obter insights sobre o ambiente que pretendo desenvolver;
- Iniciar a implementação do ambiente proposto.

Observação: [caso precise fazer alguma observação, de qualquer "natureza"]

Cheguei a pesquisar, como sugerido pela banca no último Gate, frameworks para trabalhar com Sistemas Multiagentes sem Reforço. Um exemplo encontrado foi o Mesa. Porém decidi que não era factível trabalhar com essas bibliotecas dentro do cenário de Aprendizado por Reforço pois elas são excelentes para modelar sistemas com mais de um agente e simulá-los, porém não existe uma maneira fácil de implementar esses ambientes como MDPs, o que é vital no cenário almejado.

ACEITE DA ENTREGA:

CEDRIC LUIZ DE CARVALHO: Go!

Artigos de Aprendizado por Reforço Multiagente

Este documento é um compilado de resumos de artigos na área de Aprendizado por Reforço Multiagente.

Artigo 1: A Comprehensive Survey of Multiagent Reinforcement Learning (Buşoniu et al, 2008);

O artigo fornece uma extensa revisão sobre as técnicas de Aprendizado por Reforço em Sistemas Multiagente (MARL). Esses sistemas consistem em múltiplos agentes autônomos que interagem em um ambiente compartilhado, com aplicações em diversas áreas, como robótica, controle distribuído, telecomunicações e economia. Devido à complexidade dos ambientes em que operam, os agentes frequentemente precisam aprender comportamentos novos por meio da interação, em vez de seguirem regras pré-programadas.

O principal desafio do MARL é definir claramente o objetivo do aprendizado multiagente. O artigo identifica dois focos principais de pesquisa:

1. **Estabilidade** das dinâmicas de aprendizado dos agentes, o que se refere à convergência para uma política estável.
2. **Adaptação** ao comportamento dinâmico de outros agentes, ou seja, como os agentes ajustam suas estratégias à medida que os outros também aprendem e mudam de comportamento.

Dessa forma, os algoritmos de MARL podem ser projetados para enfatizar a estabilidade, a adaptação ou uma combinação de ambas, dependendo do contexto da tarefa.

O artigo classifica os problemas em três grandes categorias, conforme o tipo de interação entre os agentes:

- **Totalmente cooperativas**: Todos os agentes compartilham um objetivo comum, como em equipes de robôs que colaboram para completar uma tarefa.
- **Totalmente competitivas**: Aqui, os interesses dos agentes são completamente opostos, como em jogos de soma zero.
- **Tarefas mistas**: Uma combinação de cooperação e competição, comum em cenários onde os agentes competem por recursos limitados, mas também precisam colaborar em algumas circunstâncias.

O artigo destaca vários desafios únicos ao MARL:

- **Dimensionalidade exponencial**: Com o aumento no número de agentes, as combinações possíveis de estados e ações aumentam exponencialmente, dificultando a solução eficiente dos problemas.
- **Não-estacionariedade**: Como os agentes aprendem simultaneamente, o ambiente torna-se dinâmico, mudando conforme os comportamentos dos outros agentes se ajustam, o que invalida a maioria das garantias de convergência dos algoritmos de aprendizado de agente único.
- **Coordenação**: Muitas vezes, os agentes precisam coordenar suas ações para alcançar um comportamento conjunto eficaz, especialmente em tarefas cooperativas.

O artigo revisa uma seleção representativa de algoritmos de MARL, agrupando-os de acordo com os objetivos de aprendizado e a natureza das tarefas que abordam. Alguns exemplos de algoritmos discutidos incluem:

- **Q-learning cooperativo**: Para tarefas totalmente cooperativas, onde os agentes aprendem uma política conjunta.
- **Minimax-Q**: Para tarefas totalmente competitivas, baseando-se no princípio do minimax, que visa maximizar o ganho sob a pior situação possível.
- **Algoritmos para tarefas mistas**: Aqui, são exploradas estratégias de equilíbrio, como o equilíbrio de Nash, que define uma situação onde nenhum agente pode melhorar sua recompensa mudando unilateralmente sua estratégia.

O MARL oferece várias vantagens, como:

- **Computação paralela**: Com múltiplos agentes agindo simultaneamente, é possível acelerar a solução de problemas.
- **Compartilhamento de experiências**: Agentes podem se beneficiar mutuamente ao compartilhar conhecimento sobre o ambiente, por meio de comunicação ou imitação.
- **Robustez**: Sistemas multiagente tendem a ser mais robustos, pois, em caso de falha de um agente, outros podem compensar suas funções.

Artigo 2: Multi-Agent Reinforcement Learning: A Review of Challenges and Applications (Canese et al, 2021)

O artigo **"Multi-Agent Reinforcement Learning: A Review of Challenges and Applications"** aborda as complexidades e desafios do aprendizado por reforço multiagente (MARL), uma extensão do aprendizado por reforço (RL) que lida com múltiplos agentes em ambientes colaborativos ou competitivos. Enquanto o RL tradicional envolve um único agente que aprende a interagir com o ambiente por meio de tentativa e erro, o MARL

adiciona a necessidade de coordenação entre agentes, tornando o ambiente dinâmico e não-estacionário. Esse ambiente mutável é um dos principais desafios, uma vez que as ações de um agente afetam o estado do ambiente e, por consequência, as recompensas recebidas pelos outros agentes. Outro desafio importante é a escalabilidade, já que o número de combinações de ações aumenta exponencialmente à medida que o número de agentes cresce, exigindo mais poder computacional. Além disso, muitos cenários práticos envolvem ambientes parcialmente observáveis, onde os agentes têm informações limitadas sobre o estado global.

Para abordar esses problemas, o artigo apresenta e analisa diversos algoritmos desenvolvidos para MARL. Esses algoritmos buscam solucionar questões como a não-estacionariedade, a escalabilidade e a observabilidade parcial:

- **Q-Learning Histerético**: Esse algoritmo ajusta a taxa de aprendizado de forma diferente para atualizações que aumentam ou diminuem os valores de Q, permitindo que os agentes ignorem temporariamente as más escolhas feitas por outros agentes, o que é útil em ambientes colaborativos.
- **REINFORCE**: Um método de gradiente de política, no qual a política é otimizada diretamente sem a necessidade de estimar as funções de valor. Apesar de sua simplicidade, o algoritmo sofre com variância alta na estimativa dos gradientes, o que pode dificultar a eficiência em problemas complexos.
- **A3C (Asynchronous Advantage Actor-Critic)**: Esse algoritmo combina elementos de métodos baseados em valor e política, onde múltiplas instâncias do ambiente são executadas de forma assíncrona para acelerar o aprendizado. Ele lida melhor com a escalabilidade e a convergência em ambientes dinâmicos.

O artigo também discute as aplicações do MARL em áreas como robótica, gestão de energia e controle de tráfego. Em sistemas robóticos, por exemplo, MARL é utilizado para coordenar drones em tarefas de vigilância e busca, onde os agentes precisam colaborar para cobrir áreas específicas enquanto evitam colisões. Na gestão de energia, algoritmos MARL permitem a coordenação entre edifícios e veículos elétricos em microrredes, otimizando o uso de energia e reduzindo custos operacionais. No controle de tráfego, os algoritmos são aplicados para gerenciar sistemas de semáforos ou veículos autônomos, colaborando para reduzir congestionamentos e melhorar a fluidez no trânsito.

Vários ambientes de simulação são utilizados para testar a eficácia desses algoritmos, como o Starcraft II, que simula cenários de combate cooperativo com múltiplas unidades heterogêneas, e o MuJoCo, um simulador de física frequentemente empregado para testar algoritmos em sistemas de controle robótico. O artigo ressalta que, apesar dos avanços significativos, os algoritmos de MARL ainda enfrentam desafios práticos, como a

necessidade de melhor eficiência e escalabilidade quando aplicados a cenários reais mais complexos e com muitos agentes.

APÊNDICE 4

Termo de Aceite de Entrega

Objetivo deste documento

Este documento faz parte do Processo da disciplina Residência em IA e tem como objetivo formalizar o aceite da entrega considerando o planejado e o realizado para o período.

Data da Reunião ("gate") de aprovação: 17 de out. de 2024

Participantes da Entrega [matriculados em Residência em IA]:

ANDRÉ LUIS ARAÚJO DE SOUZA

Entrega: [descrever a ENTREGA: requisitos e produtos gerados: links para textos, códigos, vídeos etc.]

Durante a Semana, as seguintes atividades foram realizadas no contexto da especialização em Aprendizado por Reforço:

- Implementação de uma primeira versão do simulador, que não envolve nenhuma biblioteca de RL. Esse código consiste em uma simulação simples seguindo os critérios apresentados na semana passada, onde os agentes são puramente aleatórios:
  - https://github.com/4ndr3lu15/TradeBot-Arena
- Experimentos com frameworks multiagentes que fracassaram:
  - https://github.com/4ndr3lu15/MARL-Studies
  - A ideia era implementar alguns ambientes multiagente e rodar um algoritmo útil para aquele ambiente, mas não foi possível devido a uma grande quantidade de erros que surgiram no caminho. O melhor resultado alcançado foi treinar uma única iteração de um algoritmo para um ambiente mono agente.
- Seleção de dois trabalhos anteriores:
  - https://ieeexplore.ieee.org/document/10650035
  - https://dl.acm.org/doi/10.1145/3490354.3494372

Planejamento: [descrever o que pretende fazer para realizar a próxima ENTREGA]

Os próximos passos planejados consistem em:

- Continuar experimentos de treino de algoritmos em ambientes multiagentes utilizando os frameworks estudados;
- Testar mais frameworks para tentar criar um ambiente estável de experimentação, como:
  - Mava
  - epymarl
  - acme
- Estudar os trabalhos selecionados.

Observação: [caso precise fazer alguma observação, de qualquer "natureza"]

ACEITE DA ENTREGA:

CEDRIC LUIZ DE CARVALHO: Go!

Instituto de Informática (INF) - UFG
Alameda Palmeiras, Quadra D, Câmpus Samambaia
Fone: (62) 3521-1181 / Fax: (62) 3521-1182

Termo de Aceite de Entrega

Objetivo deste documento

Este documento faz parte do Processo da disciplina Residência em IA e tem como objetivo formalizar o aceite da entrega considerando o planejado e o realizado para o período.

Data da Reunião ("gate") de aprovação: 31 de out. de 2024

Participantes da Entrega [matriculados em Residência em IA]:

ANDRÉ LUIS ARAÚJO DE SOUZA

Entrega: [descrever a ENTREGA: requisitos e produtos gerados: links para textos, códigos, vídeos etc.]

Durante a Semana, as seguintes atividades foram realizadas no contexto da especialização em Aprendizado por Reforço:

- Experimentos com frameworks multiagentes que funcionaram:
  - https://github.com/4ndr3lu15/MARL-Studies;
  - CartPole_v1: problema do pêndulo invertido;
  - tictactoe_v3: ambiente adversarial simples para validação de aprendizado - multiagente onde apenas um aprende;
  - pistonball_v6: ambiente cooperativo multiagente.
- Leitura de dois artigos:
  - https://dl.acm.org/doi/10.1145/3490354.3494372
  - https://strategicreasoning.org/wp-content/uploads/2024/10/MarketSim_ICAIF24.pdf
  - Insights retirados sobre possíveis experimentos: os dois trabalhos mencionam tipos diferentes de agentes (MM e LT) - será que esses comportamentos podem emergir ou precisam ser pré-configurados?
- Fork e leitura parcial do código do simulador encontrado em um dos artigos:
  - https://github.com/4ndr3lu15/pymarketsim

Planejamento: [descrever o que pretende fazer para realizar a próxima ENTREGA]

Os próximos passos planejados consistem em:

- Implementar um ambiente reproduzível para os experimentos que vão ser realizados no PyMarketSimulator;
- Iniciar testes iniciais com um primeiro experimento: aprendizado de um agente contra políticas pré-programadas;
- Navegar por trabalhos na ICAIF e procurar por trabalhos semelhantes ao que eu pretendo fazer.

Observação: [caso precise fazer alguma observação, de qualquer "natureza"]

Após encontrar o simulador mencionado, decidi abandonar a implementação do zero que eu estava fazendo para facilitar minha vida.

ACEITE DA ENTREGA:

CEDRIC LUIZ DE CARVALHO: Go!

Instituto de Informática (INF) - UFG
Alameda Palmeiras, Quadra D, Câmpus Samambaia
Fone: (62) 3521-1181 / Fax: (62) 3521-1182

APÊNDICE 5

Termo de Aceite de Entrega

Objetivo deste documento

Este documento faz parte do Processo da disciplina Residência em IA e tem como objetivo formalizar o aceite da entrega considerando o planejado e o realizado para o período.

Data da Reunião ("gate") de aprovação: 7 de nov. de 2024

Participantes da Entrega [matriculados em Residência em IA]:

ANDRÉ LUIS ARAÚJO DE SOUZA

Entrega: [descrever a ENTREGA: requisitos e produtos gerados: links para textos, códigos, vídeos etc.]

Durante essa Semana, foram realizados os seguintes avanços:

- Implementação de um ambiente reprodutível através de container Docker para realizar os experimentos;
- Correção de bugs em instanciação de bibliotecas no código;
- Testes de simulação com agentes ZI (Zero Intelligence);
- Repositório: https://github.com/4ndr3lu15/pymarketsim/tree/master

Além dos avanços, também aconteceu o seguinte contratempo:

- Teste de implementação do ambiente gym do simulador para agentes MM;
- Teste de treino com ambientes MM.

Planejamento: [descrever o que pretende fazer para realizar a próxima ENTREGA]

Como planejamento para próxima semana:

- Corrigir os bugs e criar, pelo menos, um ambiente gym do simulador que funcione.

Observação: [caso precise fazer alguma observação, de qualquer "natureza"]

ACEITE DA ENTREGA:

CEDRIC LUIZ DE CARVALHO: Go!

Termo de Aceite de Entrega

Objetivo deste documento

Este documento faz parte do Processo da disciplina Residência em IA e tem como objetivo formalizar o aceite da entrega considerando o planejado e o realizado para o período.

Data da Reunião ("gate") de aprovação: 13 de nov. de 2024

Participantes da Entrega [matriculados em Residência em IA]:

ANDRÉ LUIS ARAÚJO DE SOUZA

Entrega: [descrever a ENTREGA: requisitos e produtos gerados: links para textos, códigos, vídeos etc.]

Durante essa Semana, o trabalho consistiu em tentar debugar o código do simulador PyMarketSim para conseguir rodar um treinamento de Aprendizado por Reforço.

Para começar, usei como base um exemplo encontrado no repositório original. O problema inicial se dava quando uma função tentava criar um tensor de dimensão negativa. Investigando mais a fundo encontrei o código dessa função e debugei: o tensor que estava tentando ser criado era um vetor de n entradas, onde n era a diferença do próximo timestamp para o timestamp anterior. Acontece que o timestamp anterior acabava recebendo um valor muito maior que o atual após a chamada de uma função "run_until_next_MM_arrival()". Tentei refatorar o código para executar sem essa função - mas acabei obtendo um erro diferente: "TypeError: cannot unpack non-iterable NoneType object". Lendo mais a fundo o código percebi que era essencial instanciar um agente "MM" para rodar a simulação, e sem ele o espaço de observações não conseguiria ser formatado de uma maneira que o código continuasse funcionando. Tentando entender melhor o comportamento desse agente, precisei ir mais a fundo no código e me deparei com uma barreira: eu não consigo ter um entendimento sólido do código pois não consigo compreender a lógica por trás do mesmo - não tenho conhecimento em Simulações de Mercado Financeiro o suficiente para entender o Porquê de o código ter sido escrito como foi (como, por exemplo, qual a natureza do MM qual a maneira correta de fazer as chamadas de função desse agente).

Planejamento: [descrever o que pretende fazer para realizar a próxima ENTREGA]

Depois desse estudo e de me deparar com a barreira mencionada, decidi que vou pivotar a aplicação escolhida, uma vez que eu sairia muito do foco. Estou me propondo a virar um especialista em Aprendizado por Reforço, e não um especialista em Simulações de Mercado Financeiro.

Portanto dediquei um tempo estudando alternativas e acabei decidindo que vou continuar em um dos dois caminhos:

- Utilizar o ambiente Neuron Poker, que encontrei após pesquisar ambientes que poderiam ser úteis para realizar os estudos de Self-Play em ambientes Multiagentes que eu queria fazer no contexto de Mercado Financeiro.
- Continuação do trabalho desenvolvido na equipe SSL (Small Size League) do Pequi Mecânico da

qual faço parte. Lá temos um modelo que aprende a jogar futebol de robô em um ambiente simulado utilizando o Self-Play. O problema é que esse modelo ainda está longe de ter um desempenho minimamente aceitável para competir. Uma proposta de melhoria seria treinar um modelo de behavioral-cloning a partir de experiências coletadas de controladores humanos e começar o treinamento de Reforço a partir desse modelo.

Como próximos passos é analisar melhor esses dois caminhos, decidir um até o fim dessa semana e começar os primeiros experimentos ainda antes do próximo Gate.

Observação: [caso precise fazer alguma observação, de qualquer "natureza"]

ACEITE DA ENTREGA:

CEDRIC LUIZ DE CARVALHO: Go!

Termo de Aceite de Entrega

Objetivo deste documento

Este documento faz parte do Processo da disciplina Residência em IA e tem como objetivo formalizar o aceite da entrega considerando o planejado e o realizado para o período.

Data da Reunião ("gate") de aprovação: 27 de nov. de 2024

Participantes da Entrega [matriculados em Residência em IA]:

ANDRÉ LUIS ARAÚJO DE SOUZA

Entrega: [descrever a ENTREGA: requisitos e produtos gerados: links para textos, códigos, vídeos etc.]

Recapitulando:

- Estava trabalhando com um simulador de mercado financeiro para aplicar Aprendizado por Reforço Multiagente;
- O ambiente não estava funcionando corretamente para realizar os treinamentos;
- Tentei debugar e corrigir os erros, mas sem sucesso;
- Decidi pivotar para um ambiente de poker.

Durante essa Semana:

- Leitura do paper: https://alexkashi.com/resources/projects/MIT-CSM-final-paper.pdf;
- Leitura e estudo do código do repositório de referência;
  - Possui políticas pré-implementadas (aleatória e "equity");
- Ajustes e testes iniciais;
- Implementação do DQN usando torch;
- Pode ser verificado nesse commit.

Planejamento: [descrever o que pretende fazer para realizar a próxima ENTREGA]

Para a próxima Semana o planejamento consiste em realizar um treinamento do DQN usando self-play e avaliá-lo contra uma política aleatória e uma política "equity".

Observação: [caso precise fazer alguma observação, de qualquer "natureza"]

ACEITE DA ENTREGA:

LEONARDO ANTÔNIO ALVES: Em análise!

Instituto de Informática (INF) - UFG
Alameda Palmeiras, Quadra D, Câmpus Samambaia
Fone: (62) 3521-1181 / Fax: (62) 3521-1182

APÊNDICE 6

Termo de Aceite de Entrega

Objetivo deste documento

Este documento faz parte do Processo da disciplina Residência em IA e tem como objetivo formalizar o aceite da entrega considerando o planejado e o realizado para o período.

Data da Reunião ("gate") de aprovação: 5 de dez. de 2024

Participantes da Entrega [matriculados em Residência em IA]:

ANDRÉ LUIS ARAÚJO DE SOUZA

Entrega: [descrever a ENTREGA: requisitos e produtos gerados: links para textos, códigos, vídeos etc.]

Durante a Semana, estive focado em escrever um experimento para rodar um treinamento Self-play e um teste para avaliar o modelo resultante contra dois agentes aleatórios em uma mesa de Poker de 3 jogadores.

A implementação do algoritmo de treinamento usou como base o DQN que eu já havia implementado semana passada. Porém para conseguir implementar um módulo de treino, eu precisava separar o agente do treino, uma vez que eu teria vários agentes que seriam atualizados no treinamento. O código pode ser observado através dos últimos 7 commits disponíveis em https://github.com/4ndr3lu15/neuron_poker/commits/master/.

Após a implementação e correção de alguns bugs, consegui rodar 2 treinamentos, num primeiro momento: um DQN Self-play, e um DQN jogando contra 2 agentes aleatórios. Depois repeti o segundo treino por mais épocas. Registrei os dados em um Tensorboard que podem ser acessados através do seguinte diretório, ou do seguinte Google Colab:

- Experiment Output Data
- Gráficos.ipynb

Após uma análise dos dados de experimento consegui observar o seguinte:

- Provavelmente existe um erro de implementação no cálculo da recompensa para o Self-play, pois ele está dando valores que não são razoáveis;
- O gráfico da Recompensa de todos os experimentos é decrescente;
- O gráfico dos Vencedores aparentemente mostra que o Self-play estava ganhando mais consistentemente de suas cópias passadas;
- O Self-play acaba produzindo episódios muito longos no treinamento: os dois tipos de treinamento geraram um comportamento "fujão".
- Esse comportamento pode ser verificado nos testes, onde ambos os agentes tiveram 1% e 0% de winrate respectivamente.

A análise dos resultados pode ser encontrada em mais detalhes no seguinte documento:

- Análise de Resultados

Próximos passos:
- Elaborar uma função de recompensa melhor;
- Adicionar sistema de ligas no Self-play;
- EGTA (Empirical Game-Theoretic Analysis);
- Acelerar o treinamento através de ambientes paralelos;
- Algoritmos mais robustos como A3C e PPO.

Planejamento: **[descrever o que pretende fazer para realizar a próxima ENTREGA]**

Observação: [caso precise fazer alguma observação, de qualquer "natureza"]

ACEITE DA ENTREGA:

CEDRIC LUIZ DE CARVALHO: Go!

Análise de Resultados

Experimentos:

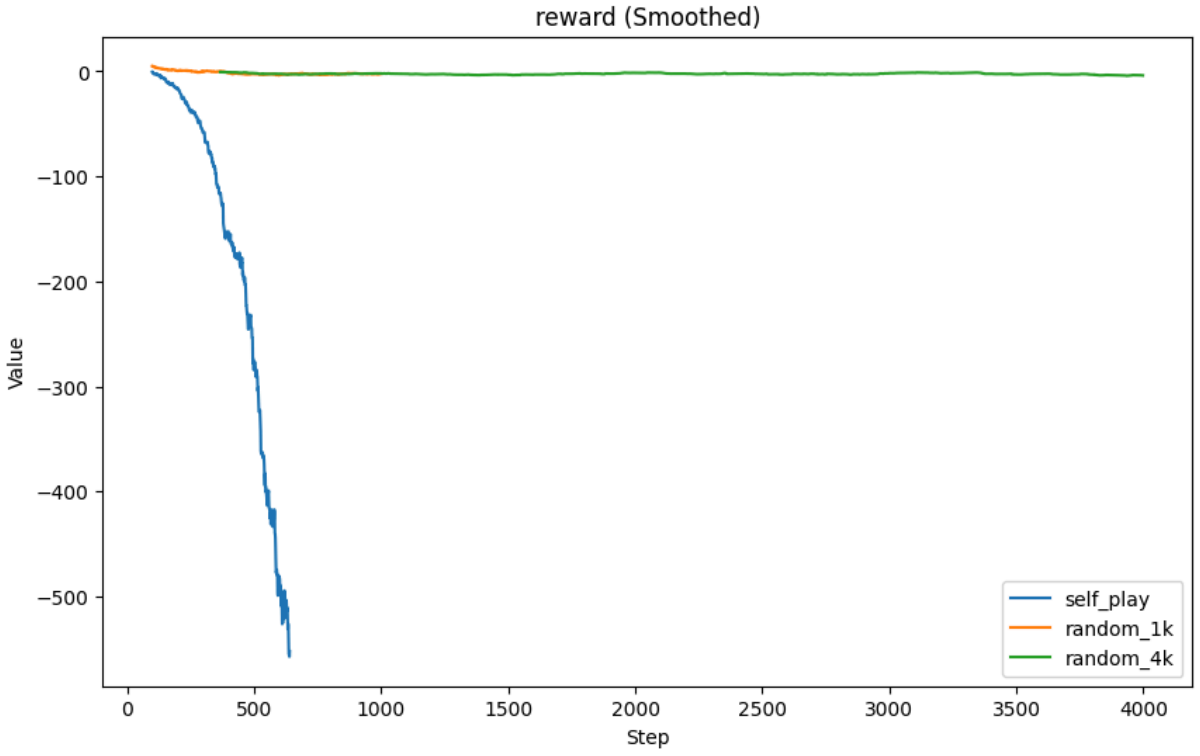
3 unidades de experimentos foram realizados:

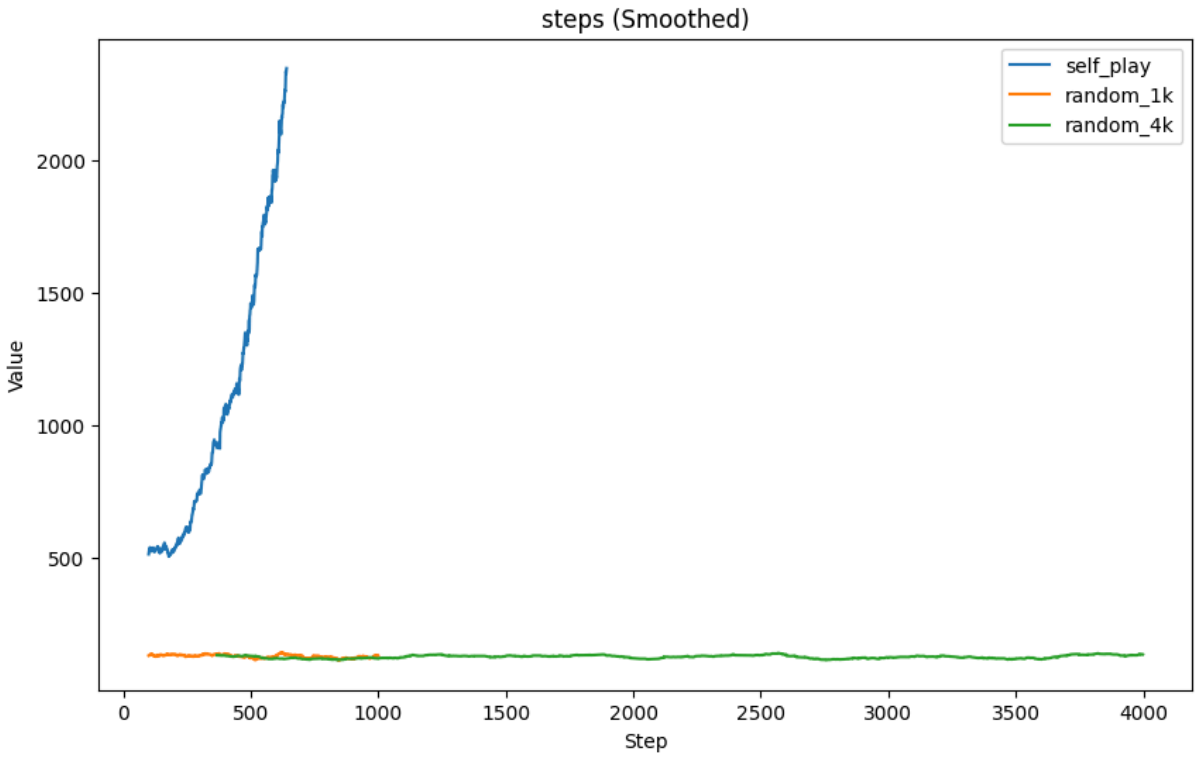
- DQN Self-play: DQN trinado contra duas cópias temporárias da própria política por aproximadamente 600 episódios;
- DQN vs Random - 1k: DQN treinado contra duas políticas Random, por 1000 episódios;
- DQN vs Random - 4k: DQN treinado contra duas políticas Random, por 4000 episódios.

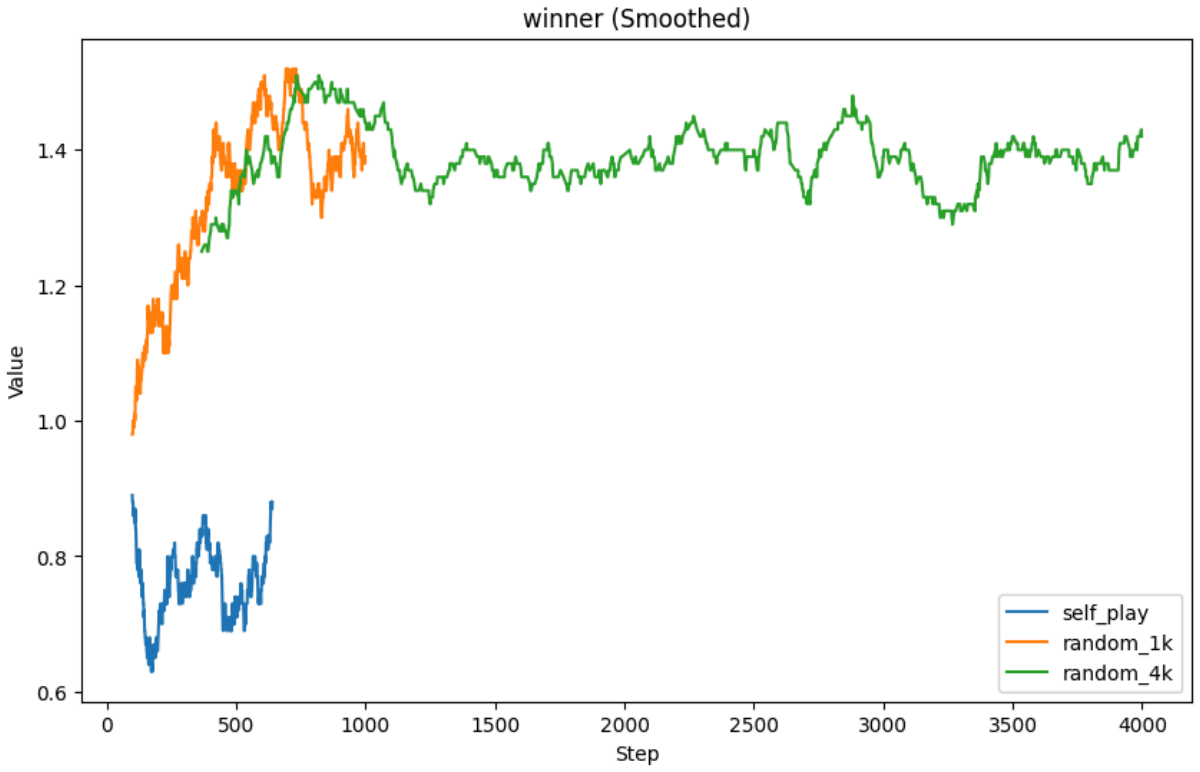
Para cada experimento, foram coletados 3 métricas:

- Recompensa: objetivo a ser maximizado - função definida por usuário;
- Passos por episódio: para medir o quanto demora cada episódio;
- Vencedor: para medir qual agente ganhou cada episódio.

Métricas:


reward (Smoothed)
Value
Step
0
−100
−200
−300
−400
−500
0
500
1000
1500
2000
2500
3000
3500
4000
self_play
random_1k
random_4k


steps (Smoothed)
self_play
random_1k
random_4k
Value
Step
0
500
1000
1500
2000
2500
3000
3500
4000


winner (Smoothed)
self_play
random_1k
random_4k
Value
Step
0.6
0.8
1.0
1.2
1.4
0
500
1000
1500
2000
2500
3000
3500
4000

Com base nos gráficos conseguimos observar algumas coisas:

- A recompensa do Self-play está caindo de uma maneira brutal: isso indica algum erro de implementação que não foi encontrado até a escrita deste documento;
- O número de passos episódios do Self-play tende a aumentar: isso foi verificado pelos resultados dos testes realizados e vai ser discutido na próxima seção;
- Os resultados dos experimentos com Random parecem não exibir algum comportamento que se destaca, a Recompensa não parece subir, mas também não cai drasticamente como no Self-play. (Esses gráficos podem ser encontrados em maior detalhe em Gráficos.ipynb);
- O número de passos por episódio tende a ficar estável nos experimentos Random.
- O gráfico dos vencedores parece indicar que o algoritmo treinado está ganhando os episódios de maneira mais consistente que os agentes congelados, mas esse comportamento não é verificado nos testes.

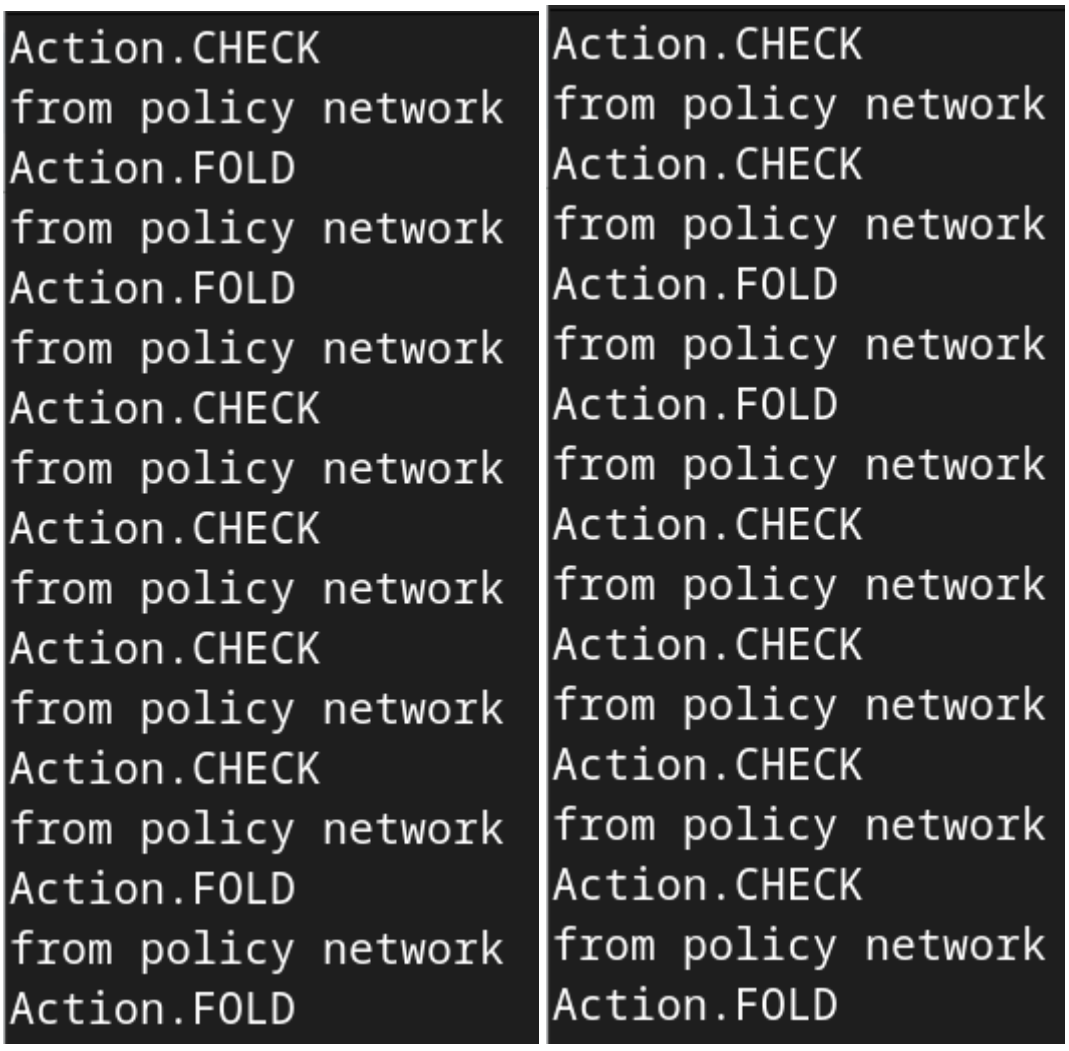
Testes

Para o teste, cada um dos algoritmos foi colocado em uma mesa com dois jogadores aleatórios e jogaram por 100 episódios. A medida observada foi a taxa de vitória de cada um:

Algoritmo	Taxa de Vitória
DQN Self-play	1%
DQN Random	0%

Observação:

Aparentemente ambos aprenderam a perder no poker. Verificando o log de ações tomadas durante a inferência, podemos ver que o agente apenas dá *check* e *fold*. Isso provavelmente está ligado à função de recompensa - se o agente foi ultra conservador ele perde menos do que se apostar alguma coisa, então ele nunca aposta, mesmo podendo ganhar apostando. Esse comportamento possivelmente emerge em um cenário com mais de dois jogadores, pois não foi observado em um dos papers de referência. Abaixo temos um print dos logs mostrando o comportamento de ambos os agentes:


Action.CHECK
from policy network
Action.FOLD
from policy network
Action.FOLD
from policy network
Action.CHECK
from policy network
Action.CHECK
from policy network
Action.CHECK
from policy network
Action.CHECK
from policy network
Action.FOLD
from policy network
Action.FOLD
Action.CHECK
from policy network
Action.CHECK
from policy network
Action.FOLD
from policy network
Action.FOLD
from policy network
Action.CHECK
from policy network
Action.CHECK
from policy network
Action.CHECK
from policy network
Action.CHECK
from policy network
Action.CHECK
from policy network
Action.FOLD