Criação de um agente capaz de jogar *Breakout* usando diferentes algoritmos de *Reinforcement Learning*

Bruno Monteiro Franzini[1], Rodrigo Souza Cardoso[2] e Rômulo Magno Rodrigues Bório[3]

***Resumo*—No presente trabalho, foi implementada uma técnica para criação e treinamento de uma rede neural, criada com auxílio da API Keras, capaz de aprender a jogar *Breakout*, um antigo jogo do Atari, por meio de *Reinforcement Learning*, com diferentes algoritmos: DQN (com 3 diferentes configurações de taxa de exploração), A2C e PPO. O ambiente do jogo foi criado com o uso da biblioteca gym, da OpenAI e as diferenças dos resultados serão discutidas ao longo das próximas seções.**

***Palavras-chave*—rede neural, *Reinforcement Learning*, *Breakout*, gym, Keras.**

I. Introdução

A inteligência artifical (IA) é caracterizada por máquinas ou sistemas que buscam ser capazes de pensar e agir de forma semelhante aos seres humanos, o que pode ser feito criando uma rede neural e a treinando. Para isso, existem diversas técnicas, cada qual fazendo mais sentido em uma aplicação diferente. Nesse sentido, para o treinamento da rede criada nesse trabalho, utilizou-se uma das técnicas mais estudadas nesse campo: o aprendizado por reforço (do inglês, Reinforcement Learning - RL). O objetivo foi o de criar um agente capaz de aprender a jogar o jogo Breakout do Atari, baseado somente nas recompensas dadas de acordo com suas decisões tomadas. As ferramentas utilizadas para a criação e treinamento desse modelo serão melhor explicadas e detalhadas nas próximas seções.

II. Fundamentação teórica

A. Python

Atualmente a linguagem *Python* é uma das mais utilizadas para aplicações de Inteligência Artificial, o que inclui tanto visão computacional quanto redes neurais. A visão computacional é o ramo da IA que se preocupa em transformar imagens em informações que podem ser tratadas e lidas por outros sistemas, simulando funções relacionadas à visão humana. Já a chamada rede neural é a estruturação de nós formando camadas conectadas, o que permite simular em *software* funções relacionadas ao cérebro humano e sua estrutura de neurônios. A Figura 1 mostra a estrutura básica de uma rede neural, com uma camada de entrada (*input*), uma de saída (*output*) e uma camada intermediária (*hidden*).

Dessa forma, existem bibliotecas específicas na linguagem *Python* que auxiliam na construção da análise. As que foram utilizadas são:

[1]bruno.franzini@ga.ita.br
[2]rodrigo.cardoso@ga.ita.br
[3]romulo.borio@ga.ita.br

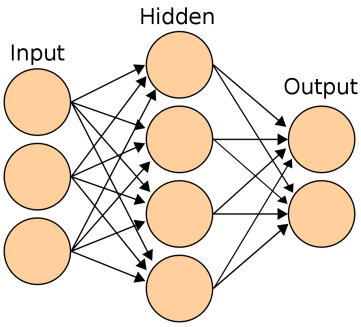

Fig. 1. A estrutura básica de uma rede neural

- Os: oferece uma forma de manipular o sistema operacional por meio do código, por exemplo, abrir arquivos, criar diretórios e arquivos temporários;
- Gym: proporciona uma grande variedade de ambientes de simulação para diferentes problemas que buscamos resolver com RL;
- Stable Baselines 3: proporciona uma implementação e treinamento facilitados de vários algoritmos de RL, como os aqui utilizados: DQN, A2C e PPO, por exemplo.
- TensorBoard: ferramenta que permite a visualização de gráficos interativos, muito útil para analisar dados.

B. Aprendizado por reforço

Um forte campo de estudo e desenvolvimento atual é o aprendizado por reforço (do inglês, Reinforcement Learning). Essa técnica é definida por uma característica principal: em vez de mostrarmos exemplos para a rede neural de tal forma que ela faça um aprendizado supervisionado, nós deixamos que essa rede (à qual frequentemente nos referimos como "agente") tome suas próprias decisões e a recompensamos (ou punimos) por isso. Ou seja, se for para o caminho certo, será recompensada, caso contrário, será punida. Esse fluxo é ilustrado na Figura 2, em que podemos ver o seguinte: o agente escolhe uma ação e a executa; com isso, o ambiente processa o estado desse agente, atualizando o estado ao qual essa ação o levou e proporcionando uma recompensa baseada nisso (se a ação foi boa ou não para alcançar o objetivo).

C. Redes neurais

O funcionamento de uma rede neural é simples: os dados são inseridos na camada de entrada, que posteriormente se comunica com as camadas mais internas. Estas são responsáveis pelo processamento da entrada por meio de um sistema de conexões ponderadas, isto é, cada nó em uma camada possui

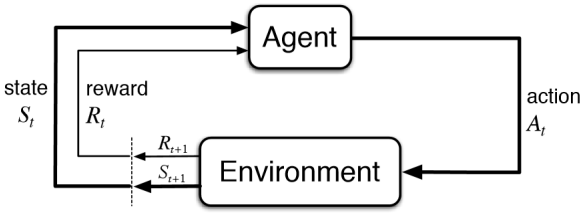

Fig. 2. Fluxo de informações durante o treinamento com RL

um peso para a entrada, formando um novo valor, sendo que, ao final de todo o processo, todos eles são somados e passados para a camada de saída, que é o resultado final.

Existem algumas classificações das redes, sendo que as chamadas de convolucionais contém cinco tipos de camadas em sua arquitetura: entrada, convolução, agrupamento (*pooling*), completamente conectada (*dense*) e saída. Elas recebem esse nome por causa da operação matemática de convolução que é efetuada na respectiva camada.

Para que uma rede funcione de forma condizente com sua finalidade dado o contexto do problema, são necessárias duas etapas: o treino e o teste. O treinamento é o passo encarregado de ajustar os pesos dos nós a partir de um processo iterativo, que no nosso caso de RL, vai ser baseado em recompensas. Por outro lado, o teste é o passo que verifica se a rede consegue atingir uma solução generalizada para o problema que deve ser resolvido.

D. Breakout

Fig. 3. Demonstração inicial do jogo Breakout

Para contextualizar o funcionamento do jogo Breakout em questão, vamos observar primeiramente a Figura 3. Nela, podemos notar que há, na parte inferior, um retângulo vermelho, que é basicamente o que podemos mexer de um lado para ou outro durante a partida. Um pouco mais acima, há um pequeno quadrado vermelho, que fica flutuando e se movimentando pela tela e o objetivo do jogo é posicionar o retângulo de modo que toda vez que o quadrado flutuante estiver caindo, nós o rebatemos de volta, para destruir os objetos coloridos na parte superior da tela.

Fig. 4. Demonstração do jogo Breakout em andamento

Na Figura 4, podemos observar como fica a tela do jogo após algum tempo de partida. Conforme mostrado, vemos que os objetos coloridos foram parcialmente destruídos e o contador avançou, aumentando a cada objeto destruído.

E. Deep Q-Network (DQN)

Um dos algoritmos de RL aqui tratados e utilizados é o Deep Q-Network (DQN), que combina a utilização de redes neurais profundas com Q-Learning para escolher a ação com maior valor Q a cada momento, ou seja, a ação ótima. Além disso, essa tomada de decisão é uma escolha que chamamos de ε-greedy, pois há uma probabilidade ε de a escolha tomada não ser com base na sua política ótima, mas sim na aleatoriedade. Isso permite que o algoritmo tenha não só *exploitation*, mas também *exploration*, fazendo com que possa sair de eventuais máximos locais, para buscar o máximo global. Além disso, outra característica importante do DQN é a memória de replay, que permite o armazenamento de experiências passadas, proporcionando um treinamento mais estável da rede neural.

F. Advantage Actor-Critic (A2C)

Outro algoritmo utilizado neste trabalho para jogar o Breakout foi o Advantage Actor-Critic (A2C). Como o próprio nome já diz, o A2C se baseia em uma arquitetura Actor-Critic (Ator e Crítico), em que temos duas redes distintas, mas que compartilham camadas iniciais, sendo uma responsável por aprender a tomar as melhores decisões (Ator) e uma responsável por estimar o valor de cada estado (Crítico), julgando assim a tomada de decisão do Ator. Diferentemente do DQN, o A2C é um algoritmo *on-policy*, o que significa que coleta experiências com base na política atualizada em cada etapa de treinamento, o que torna seu aprendizado mais eficiente e por vezes mais estável que o DQN. Além disso, para o treinamento das redes do Ator e do Crítico, é usada o gradiente ascendente para encontrar os melhores pesos de cada rede. Por fim, vale ressaltar que o A2C também utiliza o método de Monte Carlo e de diferenças temporais (TD) para estimar os valores de estados atuais e futuros, para assim encontrar a política ótima.

G. Proximal Policy Optimization (PPO)

Por fim, o último algoritmo de RL utilizado pelos autores neste trabalho foi o Proximal Policy Optimization (PPO). Esse algoritmo de RL é capaz de otimizar políticas estocásticas, ou seja, onde as ações para um dado estado não são determinísticas, podendo o agente tomar uma ação dentre várias, com diferentes probabilidades. A sua atualização de política se baseia em uma *loss function* que limita as atualizações de política (método conhecido como "clipping"), de forma que as variaçoes não sejam bruscas, ou seja, a variação é por políticas próximas, como o próprio nome diz. Isso faz com que o treinamento do PPO tenha a característica de ser suave, mais do que os outros algoritmos citados (DQN e A2C), tornando sua otimização de política mais confiável também.

III. Implementação desenvolvida

Conforme descrito anteriormente, buscamos investigar a aplicação de técnicas de Aprendizado por Reforço (Reinforcement Learning - RL) para treinar um agente inteligente para jogar o jogo Atari Breakout. Utilizamos três algoritmos distintos de RL: DQN (Deep Q-Network), PPO (Proximal Policy Optimization) e A2C (Advantage Actor-Critic). Esses algoritmos foram implementados por meio da biblioteca Stable Baselines3, uma ferramenta que fornece implementações robustas e de fácil utilização para uma variedade de algoritmos de RL.

O primeiro passo foi configurar o ambiente do jogo. Utilizamos a função make_atari_env da biblioteca Stable Baselines3, que cria um ambiente Atari e realiza uma série de pré-processamentos padrão que são úteis para o treinamento de RL. Para obter uma representação mais rica do ambiente, empilhamos quatro frames consecutivos juntos usando o VecFrameStack. Isso é particularmente útil em jogos como o Atari, onde as ações tomadas podem depender do movimento ao longo do tempo.

Em seguida, para cada um dos três algoritmos de RL, criamos uma instância do respectivo modelo. Notavelmente, para o modelo DQN, foram especificados os parâmetros de exploração (exploration_fraction, exploration_initial_eps e exploration_final_eps). Esses parâmetros desempenham um papel fundamental no controle do equilíbrio entre a exploration do ambiente pelo agente (procurar novas ações possíveis) e a exploitation (usar o conhecimento atual para tomar a melhor ação conhecida).

Os parâmetros exploration_initial_eps e exploration_final_eps definem o intervalo para o valor de epsilon, que controla a probabilidade de o agente escolher uma ação aleatória em vez de escolher a ação que acredita ser a melhor. O exploration_fraction é a fração de passos de aprendizado durante os quais o valor de epsilon será reduzido linearmente. Tendo isso em vista, conduzimos três experimentos com o DQN variando esses parâmetros para estudar como eles impactam o desempenho do agente, mantendo os outros parâmetros constantes. A Figura 5 mostra justamente o comportamento da 'exploration_rate' durante o treinamento para três diferentes possibilidades de utilização do DQN: o "DQN-1" tem os parâmetros padrão de ε; o "DQN-2" possui um valor constante e pequeno de ε; enquanto o "DQN-3" tem seu valor de ε caindo à medida que a política vai sendo otimizada, porém essa taxa de decaimento é menor do que no DQN-1 (padrão).

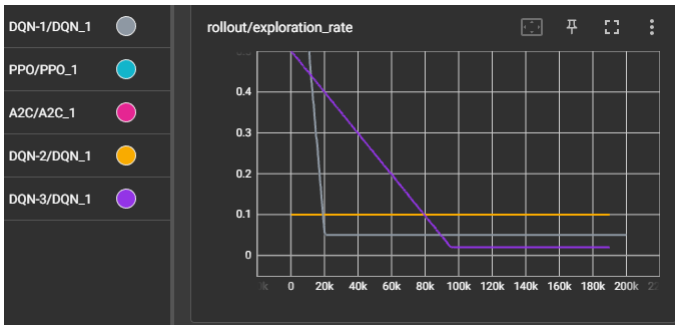

Fig. 5. Gráfico de explorarion_rate para os três diferentes DQNs testados.

Depois que os modelos foram configurados, o treinamento foi iniciado. De forma que, durante o treinamento, o agente interage com o ambiente, coleta experiências e usa essas experiências para atualizar suas políticas.

Para monitorar o progresso do treinamento e realizar análises posteriores, recorreu-se ao TensorBoard, uma ferramenta que permite visualizar diversas métricas de treinamento, como a recompensa ao longo do tempo e a perda do modelo.

Após o treinamento, os modelos foram salvos para uso posterior. Esses modelos podem ser carregados novamente para avaliação ou para continuar o treinamento a partir do ponto onde parou.

Ao ajustar os parâmetros de exploração do DQN, somos capazes de observar como diferentes estratégias de exploration/exploitation podem afetar o desempenho do agente. Essa implementação fornece uma base sólida para aprofundar a compreensão da aplicação de técnicas de RL em jogos Atari e oferece uma metodologia para explorar a influência dos parâmetros de treinamento no desempenho do agente.

IV. Resultados e Discussões

Nesta seção, discutiremos os resultados obtidos dos experimentos realizados com os algoritmos de aprendizado por reforço PPO, DQN e A2C. A análise é baseada em dois indicadores principais: 'ep_len_mean', que representa a duração média dos episódios, e 'time/fps', que é a taxa de quadros por segundo indicando a velocidade de processamento.

A Figura 6 apresenta o gráfico de 'ep_len_mean' para os três algoritmos.

No início do treinamento, o A2C apresentou uma duração média de episódio maior, seguido de perto pelo PPO e o DQN. Isso pode ser atribuído às diferenças nos métodos de exploração e na inicialização dos algoritmos. Aproximadamente nas 50k iterações, o PPO ultrapassou o A2C, possivelmente devido à sua capacidade de melhor equilibrar a exploração e a explotação.

O DQN, apesar de ter começado em um nível similar ao PPO, manteve um desempenho inferior até aproximadamente

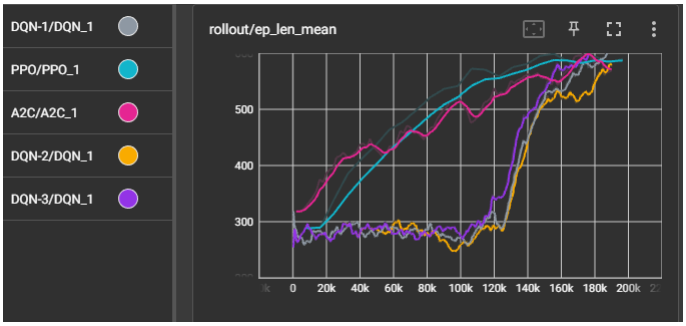

Fig. 6. Gráfico de ep_len_mean para os algoritmos PPO, DQN e A2C.

130k iterações. Isso pode ser devido à natureza da política ε-greedy do DQN que, com um valor inicial de ε alto, tende a explorar mais no início, o que pode resultar em episódios mais curtos. A partir das 130k iterações, o DQN começou a subir, sugerindo que a exploração inicial intensiva pode ter ajudado a política a convergir para uma solução melhor. Por fim, podemos destacar diferenças entre os 3 diferentes DQNs, as quais ficam mais evidentes após os 150k iterações. Conforme esperado, o DQN-2 demonstra um desempenho pior, pois sua taxa de exploração é contínua e pequena. Enquanto isso, o DQN-1, de taxa padrão, teve um desempenho intermediário e o DQN-3, com um decaimento mais lento da taxa de exploração, teve o melhor desempenho dos três. Isso pode ser explicado pela sua maior capacidade de exploração, convergindo mais lentamente, porém melhor, evitando com mais eficiência eventuais mínimos locais.

A Figura 6 apresenta o gráfico de 'time/fps' para os três algoritmos.

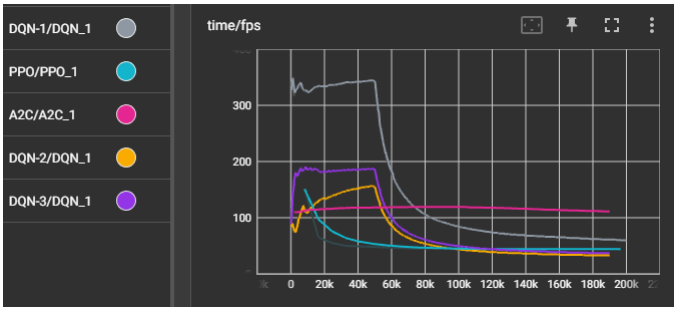

Fig. 7. Gráfico de time/fps para os algoritmos PPO, DQN e A2C.

No início do treinamento, o PPO processou os quadros mais rapidamente, seguido pelo A2C e o DQN. Isso pode ser explicado pelo fato de que o PPO, sendo um algoritmo de políticas de gradiente, pode ter uma convergência mais rápida no início. No entanto, a taxa de quadros por segundo do PPO diminuiu significativamente em torno das 50k iterações, possivelmente devido ao aumento da complexidade do problema à medida que o agente aprende.

O DQN, apesar de ter começado com a menor taxa de quadros por segundo, aumentou sua velocidade de processamento até ultrapassar o PPO e o A2C em torno das 50k iterações. Isso pode ser explicado pelo fato de que o DQN utiliza uma abordagem de aprendizado baseada em valor, que pode ser mais eficiente à medida que o tamanho do espaço de ações aumenta.

Em resumo, os resultados indicam que o equilíbrio entre exploração e explotação, bem como a eficiência de processamento, são aspectos cruciais para o sucesso do treinamento de algoritmos de aprendizado por reforço. Além disso, a escolha do algoritmo pode ter um impacto significativo no desempenho e na velocidade do treinamento, dependendo das características do ambiente e do problema em questão.

V. Conclusão

Diante do objetivo desse trabalho e dos resultados obtidos, podemos concluir que foram satisfatórios, pois foi possível observar na prática o aprendizado e execução de um algoritmo de RL e, além disso, notar as diferenças e nuances de cada um dos três aqui analisados. Ademais, observou-se, também, o impacto da taxa de exploração no aprendizado da rede neural, o que ocorreu conforme o esperado e convergiu com o que foi estudado durante o curso sobre a relação entre *exploration* e *exploitation*.