UNIVERSIDADE FEDERAL DO PARANÁ

PETERSON WAGNER KAVA DE CARVALHO

ANÁLISE DE ALGORITMOS DE APRENDIZAGEM POR REFORÇO PROFUNDO EM JOGOS DE ATARI 2600

CURITIBA PR

2021

PETERSON WAGNER KAVA DE CARVALHO

ANÁLISE DE ALGORITMOS DE APRENDIZAGEM POR REFORÇO PROFUNDO EM JOGOS DE ATARI 2600

Trabalho apresentado como requisito parcial à conclusão do Curso de Bacharelado em Ciência da Computação, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Eduardo Jaques Spinosa.

CURITIBA PR

2021

AGRADECIMENTOS

Agradeço à meus pais, Marcia Kava e Sebastião Carvalho, por todo o amor, apoio e incentivo que me deram ao longo destes anos.

Agradeço ao meu orientador, Prof. Dr. Eduardo Spinosa, pela excelente orientação e apoio durante a realização deste trabalho.

Agradeço ao meu grande companheiro Gabriel Fontes, que me fez companhia durante grande parte desta jornada. Obrigado por todo o apoio em vários momentos difíceis e por todas as tardes e noites de jogatinas e séries.

Agradeço aos meus amigos de turma e de bar, Abner Costa, Gerhard Neto, Marcelo Cardoso, Gabriel Urba, Amanda Föetsch, Jaqueline Bergling, Matheus Horstmann, Gustavo Esteves e Gabriel Sales, os quais tornaram esta jornada muito mais divertida.

Agradeço também aos meus amigos do Smite, Anderson Costa, Luiz Taffner, Anderson Ferreira, Beatriz Reis, Antão Neto e Renan Mendes. Não teria sido o mesmo sem todos vocês.

Por fim, agradeço à UFPR pelo ensino de qualidade e pela bolsa de auxílio que possibilitou todos os meus estudos na universidade.

RESUMO

Este trabalho utiliza a técnica de aprendizagem por reforço para ensinar agentes a jogarem jogos eletrônicos, sem nenhum conhecimento prévio, a partir da imagem da tela do jogo e da pontuação recebida. Para o treinamento dos agentes, são utilizados algoritmos que possuem redes neurais como parte do algoritmo de aprendizagem, incluindo *Advantage Actor-Critic*, *Sample Efficient Actor-Critic with Experience Replay*, *Deep Q-Networks* e *Proximal Policy Optimization*. O trabalho foi desenvolvido utilizando as ferramentas *Stable Baselines* que implementa os algoritmos de aprendizagem por reforço e *OpenAI Gym* que fornece os ambientes de jogos emulados de Atari 2600. Um primeiro experimento visa comparar a performance dos algoritmos nos jogos *Pong*, *Ms. Pac-Man*, *Seaquest* e *Montezuma's Revenge*, onde observou-se que em três dos quatro jogos os agentes realizaram algum aprendizado no ambiente, sendo que em um deles os agentes ultrapassaram a média de pontuação humana. Um segundo experimento de transferência de aprendizado é feito utilizando os jogos *Space Invaders* e *Demon Attack*, treinando os agentes primeiro em um jogo e em seguida em outro, afim de observar algum aproveitamento de conhecimento. Entretanto, os resultados deste último experimento indicam que não houve melhora significativa nas pontuações ao utilizar esta técnica.

Palavras-chave: Aprendizagem por Reforço. Inteligência Artificial. Jogos Eletrônicos. A2C. ACER. DQN. PPO.

ABSTRACT

This work uses reinforcement learning technique to teach agents to play electronic games, without any prior knowledge, based on the game's screen image and the score received. For agent training, algorithms that have neural networks as part of the learning algorithm are used, including Advantage Actor-Critic, Sample Efficient Actor-Critic with Experience Replay, Deep Q-Networks and Proximal Policy Optimization. The work was developed using the frameworks Stable Baselines that implements the reinforcement learning algorithms and OpenAI Gym that provides the Atari 2600 emulated game environments. A first experiment aims to compare the performance of the algorithms in the games Pong, Ms. Pac-Man, Seaquest and Montezuma's Revenge, where it was observed that in three of the four games the agents performed some learning in the environment, and in one of them the agents exceeded the average human score. A second experiment in transfer learning is performed using the games Space Invaders and Demon Attack, training agents first in one game and then in another, in order to observe some use of knowledge. However, the results of this last experiment indicate that there was no significant improvement in scores when using this technique.

Keywords: Reinforcement Learning. Artificial Intelligence. Electronic Games. A2C. ACER. DQN. PPO.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ACRÔNIMOS

A2C	*Advantage Actor Critic*
A3C	*Asynchronous Advantage Actor Critic*
ACER	*Sample Efficient Actor-Critic with Experience Replay*
DQN	*Deep Q-Networks*
DRL	*Deep Reinforcement Learning*
MDP	*Markov Decision Process*
PPO	*Proximal Policy Optimization*
RL	*Reinforcement Learning*
TRPO	*Trust Region Policy Optimization*

LISTA DE SÍMBOLOS

α	Fator de aprendizagem
β	Fator de aprendizagem
t	Unidade discreta de tempo
a	Ação realizada por um agente
s	Estado do ambiente
Q	Função de valor ação-estado
G	Recompensa cumulativa
L	Função de erro
E	Esperança matemática
π	Política de um agente
V	Função de valor de um estado
δ	Erro de diferença temporal
A	Função de vantagem
γ	Fator de desconto sobre recompensas futuras
r	Recompensa recebida em uma transição do ambiente
ϵ	Probabilidade de realizar uma ação aleatória
ω	Amostragem de importância

SUMÁRIO

1 INTRODUÇÃO

Graças aos avanços na área de Inteligência Artificial (IA), hoje podemos realizar tarefas que seriam impossíveis anos atrás. Assistentes virtuais, como a Alexa da Amazon, são capazes de conversar com seus usuários e auxiliá-los em tarefas como agendamento de consultas e controle das luzes da casa. Lojas de *e-commerce* contam com sistemas de recomendação para identificar os gostos dos usuários e oferecer produtos adequados. Carros autônomos são capazes de dirigir sozinhos, tomando decisões a partir de suas observações sobre o trânsito e os seus arredores.

Uma subárea promissora da IA é a Aprendizagem por Reforço, que funciona de maneira semelhante ao condicionamento operante proposto por Skinner (1938), que é um tipo de aprendizagem por associação em que um comportamento é estimulado com recompensas ou inibido com punições. Um cachorro, por exemplo, pode ter seu comportamento alterado ao receber comida ou carinho como *feedback* positivo para toda vez que ele faz um truque, ou uma advertência verbal ou negação de carinho como *feedback* negativo para toda vez que fizer algo indesejado.

Recentemente, vimos um grande crescimento nesta área devido aos avanços de *hardware* nas *GPUs* (*Graphics Processing Unit*), que são especialmente úteis em processamentos com grande número de repetições de cálculos devido ao seu alto nível de paralelismo. Isto possibilitou o uso de redes neurais, algoritmo que imita o funcionamento do cérebro humano para reconhecer padrões em dados que passou a ser utilizada em diversas áreas da IA, inclusive na Aprendizagem por Reforço.

Jogos eletrônicos são frequentemente utilizados para desenvolver e avaliar algoritmos de aprendizagem por reforço, visto que ambientes simulados são mais baratos que ambientes físicos no processo de treinamento e avaliação de performance dos algoritmos. Além disso, jogos proporcionam um ambiente livre de influência externa e com reprodutibilidade dos testes, perfeito para o desenvolvimento de modelos de IA.

Em 1997, o supercomputador *DeepBlue* da IBM ganhou uma partida de xadrez contra o reconhecido jogador russo Garry Kasparov. Em 2016, a IA *Alpha Go* derrotou o jogador profissional Lee Sedol em uma partida de Go, o que até então não havia acontecido devido a alta complexidade do jogo pela sua imensa árvore de probabilidades. Mais recentemente, em 2019 a IA *OpenAI Five* ganhou um melhor de três de um time profissional do jogo *Dota 2*, um jogo altamente competitivo com múltiplos objetivos que requer cooperação e colaboração entre os 5 jogadores. A partir destes eventos, fica claro que a comunidade científica continuará tentando quebrar metas cada vez mais complexas na área de IA em jogos.

Vários algoritmos modernos de aprendizagem por reforço, como o *Advantage Actor-Critic* e o *Deep Q-Networks*, foram propostos utilizando jogos de Atari 2600 como um dos tipos de ambiente de validação, utilizando o *framework Arcade Learning Environment* que oferece ferramentas de emulação e métricas próprias para estudos de IA. O treinamento de modelos nos 59 jogos do catálogo do Atari 2600 permitem aos pesquisadores avaliarem os algoritmos em diversos tipos de cenários e sistemas de recompensas. O grande número de jogos permite uma avaliação da capacidade de generalização na aprendizagem dos modelos, o que é útil também na resolução de problemas do mundo real, já que diversas atividades presentes em jogos possuem tarefas comparáveis as do mundo real.

1.1 OBJETIVOS

Criar agentes de aprendizagem por reforço capazes de aprender a jogar jogos eletrônicos distintos a partir de informações visuais da tela e recompensas obtidas do próprio *score* do jogo, sem demonstração ou interferência humana.

Serão escolhidos múltiplos algoritmos de aprendizagem por reforço que serão comparados pelas suas performances em jogos de Atari 2600 de diferentes gêneros.

Além disso, será feito um experimento de transferência de aprendizado para tentar avaliar um aproveitamento de conhecimento no treinamento de um jogo utilizando um mesmo agente já treinado em outro jogo.

1.2 ORGANIZAÇÃO DA MONOGRAFIA

Esta monografia está organizada em cinco capítulos. No Capítulo 2, serão revisados os conceitos matemáticos e computacionais de aprendizagem por reforço utilizados neste trabalho. No Capítulo 3, será descrito o sistema proposto mostrando as ferramentas utilizadas. No Capítulo 4, serão mostrados os resultados dos experimentos realizados para comparação entre os algoritmos e do experimento de transferência de aprendizado. Finalmente, o Capítulo 5 apresenta as conclusões do trabalho e propostas para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

O capítulo apresentará conceitos de aprendizagem, aprendizagem por reforço, aprendizagem profunda, transferência de aprendizado e os algoritmos utilizados neste trabalho.

2.1 APRENDIZAGEM

Existem três tipos de aprendizagem de máquina: aprendizagem supervisionada, aprendizagem não-supervisionada e aprendizagem por reforço (Sutton e Barto, 2018).

Na aprendizagem supervisionada, o principal objetivo é mapear uma variável de entrada para uma variável de saída, sendo necessária uma base de dados rotulada, ou seja, com entradas e saídas previamente computadas. À medida que as entradas são consumidas, o modelo realiza ajustes nos seus pesos para que predição da saída fique cada vez mais próxima à da saída real. Exemplos de problemas resolvidos por aprendizagem supervisionada incluem classificação de imagens e análises preditivas para projeções de vendas de uma empresa.

Já a aprendizagem não-supervisionada difere da supervisionada ao não depender de uma base de treinamento rotulada, possuindo apenas os dados de entrada sem os dados de saída. Assim, este tipo de aprendizagem se especializa em observar padrões nos dados de entrada e é muito usado para resolver problemas de clusterização, detecção de anomalias em um conjunto de dados e sistemas de recomendação.

Na aprendizagem por reforço o agente aprende a executar tarefas baseado em recompensas positivas ou negativas dependendo do seu desempenho em ações anteriores. Esta abordagem difere das outras por não ter uma base de dados pré-definida, sendo necessário realizar interações com o ambiente via tentativa e erro para obter recompensas.

2.2 APRENDIZAGEM POR REFORÇO

Na Aprendizagem por Reforço, em inglês *Reinforcement Learning* (RL), temos um **agente** que realiza decisões e um **ambiente** onde o agente vive, interage e recebe **recompensas**.

O agente deve repetidamente analisar o estado S_t do ambiente e realizar uma ação A_t dentre o conjunto de todas as ações disponíveis. Depois de cada escolha, o agente analisa um novo estado S_{t+1} e pode receber uma recompensa numérica R_{t+1} vinda do ambiente. O objetivo do agente é maximizar o total acumulado de recompensas depois de um determinado período de tempo. Este fluxo é ilustrado na Figura 2.1.

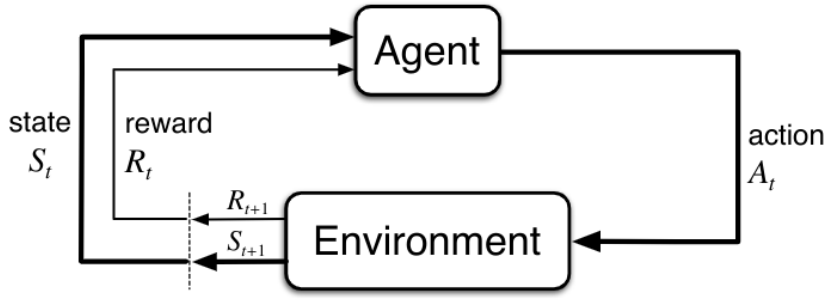

Figura 2.1: Fluxo Principal em Reinforcement Learning. (Sutton e Barto, 2018)

O **agente** é uma inteligência artificial que faz ações autonomamente em busca de um objetivo e contém dois componentes principais: um algoritmo de aprendizado e uma política.

- A **política** é um mapeamento que recebe como entrada as observações do agente sobre o estado atual do ambiente e gera como saída as ações a serem realizadas pelo agente dada uma observação.
- O **algoritmo de aprendizado** tem como objetivo achar uma política ótima que leve à recompensa acumulada máxima, atualizando-a continuamente baseado nas ações, observações e recompensas do agente.

O **ambiente** contém tudo o que está nos arredores do agente, excluindo o próprio agente. O ambiente contém três componentes: os estados, o conjunto de ações possíveis e o sistema de recompensas.

- Um **estado** é uma representação estática do ambiente em um determinado momento. O agente tem acesso às informações disponíveis do estado por meio de suas próprias observações, que podem ser incompletas dependendo do tipo de problema.
- O conjunto de **ações** possíveis de serem realizadas pelo agente em cada estado. No jogo Space Invaders, por exemplo, o agente pode escolher uma ação dentre o conjunto de ações $A = \{$*Mover para a esquerda, Mover para a direita, Atirar, Mover para esquerda e atirar, Mover para a direita e atirar, Não fazer nada*$\}$.
- O **sistema de recompensas** define o objetivo do aprendizado, como chegar a um destino ou sobreviver a ataques, enviando um sinal numérico de recompensa para o agente após cada ação.

É possível classificar ambientes em diferentes tipos como ambiente discreto ou contínuo, determinístico ou estocástico, parcial ou totalmente observável pelo agente, episódico ou não episódico e com um ou múltiplos agentes. A Tabela 2.1 mostra as principais diferenças entre cada tipo de ambiente.

Uma **recompensa** é uma métrica que diz ao agente o quão bom está o seu desempenho em cada estado do ambiente. Existem diversas funções de recompensa que podem ser utilizadas, sendo as mais comuns as funções baseadas em maximização de ganhos e minimização de perdas. A melhor função de recompensa depende fortemente do contexto de cada problema, já que não existe uma função de recompensa ideal para todo tipo de problema. Em uma partida de Doom, por exemplo, o agente pode receber uma recompensa a cada inimigo derrotado ou a cada 5 segundos de tempo que o agente permanece vivo.

O **valor** é uma estimativa da recompensa total a ser obtida a partir de um estado, seguindo uma determinada política. O valor é usado pelo agente para medir a qualidade dos estados e decidir qual a melhor ação a ser realizada em um certo momento.

2.3 PROCESSO DE DECISÃO DE MARKOV

O Processo de Decisão de Markov (*Markov Decision Process* em inglês, MDP) fornece uma estrutura matemática para resolver um problema de aprendizagem por reforço. Segundo Fenjiro e Benbrahim (2018), o MDP é definido por uma tupla $\langle S, A, R, P, \gamma \rangle$, onde:

- S é o conjunto contendo todos os estados válidos,
- A é o conjunto contendo as ações que o agente pode executar,

Tabela 2.1: Atributos de ambientes em Reinforcement Learning. Adaptado de (William John, 2010)

Atributos do Ambiente	Descrição
Determinístico ou Estocástico	Em um ambiente estocástico não é possível determinar com absoluta certeza o próximo estado após o agente executar uma ação. Já em um ambiente determinístico, como no console Atari 2600 que não possuía gerador de números aleatórios, isto é possível (Maquaire, 2020).
Único agente ou Múltiplos agentes	Um ambiente pode ter um ou múltiplos agentes que interagem com o ambiente. Em ambientes com múltiplos agentes, há a possibilidade de haver cooperatividade e competitividade entre os agentes.
Discreto ou Contínuo	Ambientes discretos têm um conjunto finito de ações disponíveis ao agente que vão de um estado a outro, como em Go e jogos da Atari 2600 que possuem um número limitado de movimentos para o agente. Ambientes contínuos possuem infinitos estados, como em agentes que controlam robôs no mundo físico, e o seu conjunto de ações possíveis possui um ou mais valores reais (OpenAI, 2018a).
Episódico ou Sequencial	Um ambiente episódico, também chamado de ambiente não-sequencial, possui um estado início e um estado final ou terminal (tipicamente com uma recompensa final). Além disso, as tarefas do agente são independentes entre si: uma ação do agente não afeta as suas ações futuras. Em um ambiente sequencial, as tarefas do agente continuam indefinidamente. Além disso, as ações do agente são relacionadas e, portanto, uma ação realizada em um determinado momento impacta as suas ações futuras.
Totalmente Observável ou Parcialmente Observável	Ambientes são totalmente observáveis quando o agente é capaz de obter todas as informações dos estados do ambiente, como em um jogo de xadrez em que o jogador consegue observar todo o tabuleiro. Ambientes são parcialmente observáveis quando o agente tem apenas uma parte das informações dos estados do ambiente, como em um jogo em primeira pessoa, em que as informações obtidas dependem do campo de visão do agente.

- R é a recompensa acumulada resultante das ações do agente,
- P é uma matriz de probabilidade de transição que descreve como o agente se move entre estados para cada ação,
- γ é o fator de desconto que faz com que as expectativas de recompensas futuras tenham um impacto menor do que as recompensas imediatas nas decisões do agente.

O MDP pode ser visualizado na Figura 2.2, onde ele é representado por um grafo com vértices representando os estados S e ações A e arestas entre eles. As arestas que têm como origem uma ação estão associadas a uma probabilidade p de transicionar o ambiente para o estado de destino da aresta, e também podem conter recompensas.

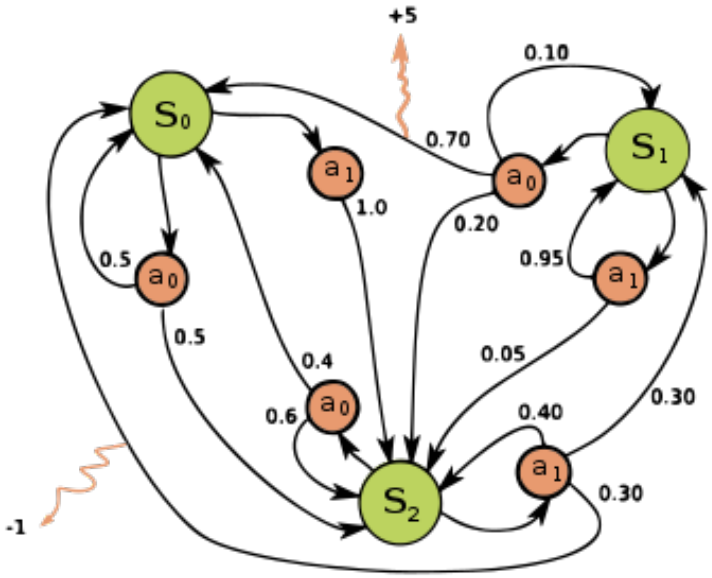

Figura 2.2: Exemplo de um MDP. (Wikipedia, 2022)

A política π é a função que define a distribuição de probabilidades de ações para estados, definindo as ações a serem tomadas pelo agente em resposta a uma observação do estado. Uma política não-determinística segue a Equação 2.1.

$$a = \pi(a|s) = P(A = a|S = s) \tag{2.1}$$

As transições de estado de um MDP satisfazem a Propriedade de Markov que define que as ações realizadas em um estado dependem somente do estado atual e não dos estados passados, como visto na Equação 2.2 (Sutton e Barto, 2018). Em outras palavras, as transições de estado de um MDP não possuem memória.

$$P(S_{t+1}|S_t) = P(S_{t+1}|S_1, \ldots, S_t) \tag{2.2}$$

O objetivo do agente é maximizar a recompensa cumulativa G_t esperada dada pela Equação 2.3

$$G_t = \gamma R_{t+1} + \gamma^2 R_{t+2} + \cdots + \gamma^{n-1} R_{t+n-1} = \sum_{k=0}^{n} \gamma^k R_{t+k+1}, \tag{2.3}$$

onde R_{t+k} é a recompensa obtida em um instante de tempo $t + k$ e γ é um fator de desconto $\in [0, 1]$.

A recompensa cumulativa esperada vem acompanhada de um fator de desconto para que as expectativas de recompensas futuras tenham um impacto menor do que as recompensas imediatas nas decisões do agente. O fator de desconto aumenta exponencialmente conforme a distância temporal do estado estimado até o estado atual, já que as estimativas tendem a ficar cada vez mais incertas.

O agente tenta aprender uma política ótima, denotada por π^*, para tomar ações que maximizem a recompensa cumulativa esperada. Para isso, os algoritmos RL fazem uma estimativa do total de recompensas futuras a serem obtidas nos estados futuros e as comparam, sendo possível escolher o melhor caminho que o agente pode tomar em um determinado momento. Esta estimativa é chamada função de valor e pode ser calculada de duas maneiras distintas:

- A Função Estado-Valor $V_\pi(s)$ estima as recompensas futuras a partir de um estado s seguindo uma política π, como na Equação 2.4.

$$V_\pi(s) = E_\pi\{\sum_{k=0}^{n} \gamma^k R_{t+k+1} | S_t = s\} \tag{2.4}$$

- A Função Ação-Valor, também chamada de função Q, $Q_\pi(s, a)$ estima as recompensas futuras ao realizar uma ação a em um estado s, seguindo uma política π, como na Equação 2.5.

$$Q_\pi(s, a) = E_\pi\{\sum_{k=0}^{n} \gamma^k R_{t+k+1} | S_t = s, A_t = a\} \tag{2.5}$$

O objetivo de um algoritmo RL é usar o agente para encontrar uma política ótima que maximize $V_\pi(S)$ ou $Q_\pi(s, a)$. A Equação de Bellman

$$G_t = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \cdots = R_t + \gamma V_\pi(s_{t+1}) \tag{2.6}$$

nos diz que em uma política ótima, o valor de um estado é igual à soma da recompensa imediata com o valor do próximo estado (Sutton e Barto, 2018). Assim, as Equações 2.4 e 2.5 podem ser transformadas nas Equações de Otimalidade de Bellman 2.7 e 2.8 para V e Q sob uma política ótima.

$$V^*_\pi(s_t) = \max_\pi [V_\pi(s_t)] = \max_{a_t} \sum_{s_{t+1}, r_t} P(S_{t+1}, r_t | s_t, a_t)[r_t + \gamma V_\pi(s_t)] \tag{2.7}$$

$$Q^*_\pi(s_t, a_t) = \max_\pi [Q_\pi(s_t, a_t)] = \sum_{s', r} P(S_{t+1}, r_t | s_t, a_t)[r_t + \gamma \max_{a_{t+1}} Q_\pi(s_t, a_{t+1})] \tag{2.8}$$

Em RL, as funções Estado-Valor ou Ação-Valor são armazenadas em memória, sendo que para cada estado s há um $V(s)$ correspondente ou para cada par ação-estado (s, a) há um $Q(s, a)$ armazenado (Fenjiro e Benbrahim, 2018). Em problemas com alta dimensionalidade de estados, o tamanho do MDP se torna muito alto para ser armazenado em memória e o processo de aprendizagem se torna lento, já que é necessário visitar cada um dos estados. Por este motivo, funções de aproximação são utilizadas em conjunto com a aprendizagem por reforço, em especial as redes neurais artificiais (que serão explicadas mais adiante) devido a sua capacidade de aprender representações de baixa dimensão e aproximação de funções.

2.4 DESAFIOS NA APRENDIZAGEM POR REFORÇO

Existem alguns desafios que devem ser levados em consideração ao modelar um problema de RL, como:

- **Exploração versus Aplicação**: Quando um agente deve tentar executar ações subótimas para explorar o ambiente e ganhar conhecimento e quando ele deve aplicar o conhecimento já adquirido e executar as melhores ações (que ele conhece)? Uma das estratégias mais comuns para resolver o problema é a abordagem ϵ-gulosa (ϵ-*greedy*), em que força o agente a explorar o ambiente escolhendo uma ação qualquer com probabilidade ϵ (parâmetro) e aplicar o seu conhecimento já adquirido com probabilidade $1 - \epsilon$ (Fenjiro e Benbrahim, 2018). O valor ϵ pode começar alto e decair ao longo do tempo para forçar a exploração do ambiente nos momentos iniciais do treinamento enquanto o agente não possui muito conhecimento, até o momento em que o agente pode apenas aplicar o conhecimento adquirido escolhendo as melhores ações.

- **Recompensas Esparsas**: Damos o nome de recompensas esparsas para situações em que o ambiente raramente dá recompensas ao agente, apenas ao final de um episódio por exemplo, o que dificulta a aprendizagem do agente já que ele não sabe exatamente qual o subconjunto de ações que causou a recompensa recebida. Uma abordagem para resolver o problema é construir uma função de recompensas contínua que complementa o sistema primário de recompensas, tornando-a menos esparsa (Fenjiro e Benbrahim, 2018).

- **Eficiência de Amostras**: Algoritmos de RL tendem a precisar de milhões de amostras de dados para aprender uma tarefa, enquanto humanos são capazes de aprender rapidamente e obter uma boa performance na mesma tarefa (Shao et al., 2019). Em jogos de Atari 2600, por exemplo, humanos conseguem em 15 minutos ultrapassar o *score* de um agente treinado por 115 horas utilizando o algoritmo DQN (que será explicado mais adiante), segundo Fenjiro e Benbrahim (2018).

2.5 REDES NEURAIS

Uma rede neural é um modelo computacional inspirado no funcionamento do cérebro humano que é muito utilizada no reconhecimento de padrões em dados.

Um neurônio é responsável por aplicar uma função de ativação sobre as entradas resultando em uma saída. A saída de um nerônio pode ser conectada as entradas dos neurônios da camada seguinte, formando um emaranhado de neurônios ligados entre si chamado de rede neural, como mostra a Figura 2.3.

A saída de cada neurônio é o resultado de uma função aplicada sobre a soma de todas as entradas multiplicadas pelos seus respectivos pesos. Uma função de degrau simples pode ser utilizada fazendo com que a saída seja igual 1 caso a soma das entradas seja maior que um valor de ativação b e 0 caso contrário, conforme a Equação 2.9, onde y é a saída do neurônio, x_i é a saída do neurônio i e w_i é o peso associado a conexão entre dois neurônios (Marvin e Seymour, 1969).

$$y = \begin{cases} 0, se \sum_{i=1}^{n} w_i x_i \leq b \\ 1, se \sum_{i=1}^{n} w_i x_i > b \end{cases} \tag{2.9}$$

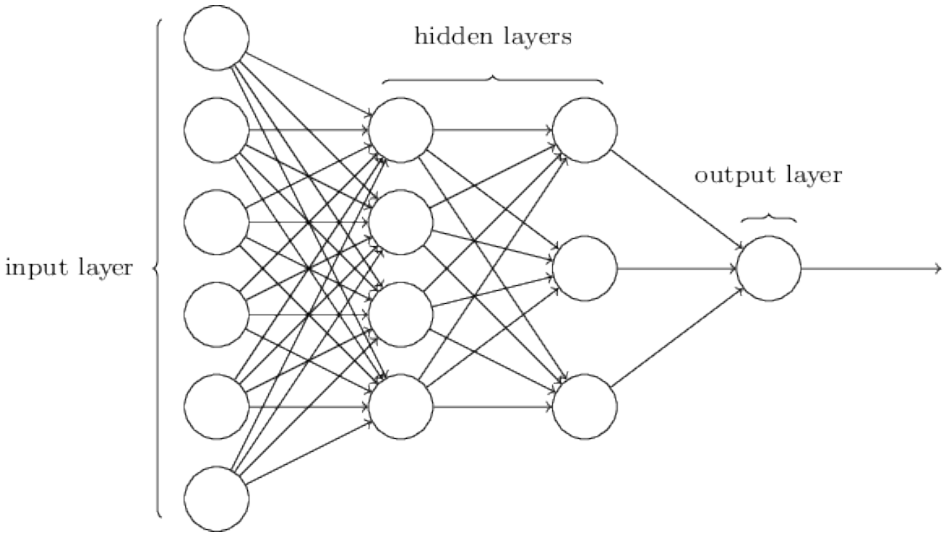

Figura 2.3: Exemplo de rede neural. Adaptado de (Nielsen, 2015)

O aprendizado do algoritmo é realizado nos valores dos pesos, que são inicializados com valores aleatórios e diferentes de zero e vão sendo atualizados a cada iteração. Após a propagação da entrada em toda a rede, é calculada a taxa de erro em cada neurônio de saída conforme a Equação 2.10, onde L é o erro, y é o valor obtido e $\hat{y}$ é o valor estimado.

$$L = (y - \hat{y})^2 \tag{2.10}$$

A partir do valor de erro, a rede é retro-propagada da última camada até a primeira subtraindo de todos os pesos a sua respectiva contribuição no erro da classificação, a fim de obter valores mais próximos ao esperado. Para isso, calcula-se a derivada parcial do erro em relação ao peso a ser atualizado para se obter a direção em que a função de erro diminui. Esta derivada parcial é multiplicada por um fator de aprendizagem e então subtraída do peso original, obtendo os novos pesos para toda a rede neural, conforme a Equação 2.11, onde w é um peso da rede neural, $\hat{w}$ é o peso atualizado, L é o erro e α é o fator de aprendizagem (Marvin e Seymour, 1969).

$$\hat{w} = w - \alpha \cdot \frac{\partial L}{\partial w} \tag{2.11}$$

Uma rede neural possui obrigatoriamente uma camada de entrada e uma camada de saída, porém pode possuir um número qualquer de camadas intermediárias. Quando uma rede possui mais de 2 camadas intermediárias, passamos a chamá-la de rede profunda (Nielsen, 2015). Cada camada em uma rede neural contribui em sua capacidade de abstração para o reconhecimento de padrões nos dados de entrada.

2.5.1 Rede Neural Convolucional

LeCun et al. (1998) propuseram uma arquitetura de rede neural mais apropriada para classificação de imagens, chamada Rede Neural Convolucional (*Convolutional Neural Network* em inglês, CNN). Esta arquitetura possui 3 tipos principais de camadas: Convolucional, Subamostragem e Totalmente Conectada como mostrado na Figura 2.4.

A camada de convolução é a camada mais importante neste tipo de rede neural. Nela, são realizadas multiplicações matriciais entre pequenas porções da imagem com filtros de mesmo tamanho, que possibilita a obtenção de informações de sub-blocos da imagem. Os filtros

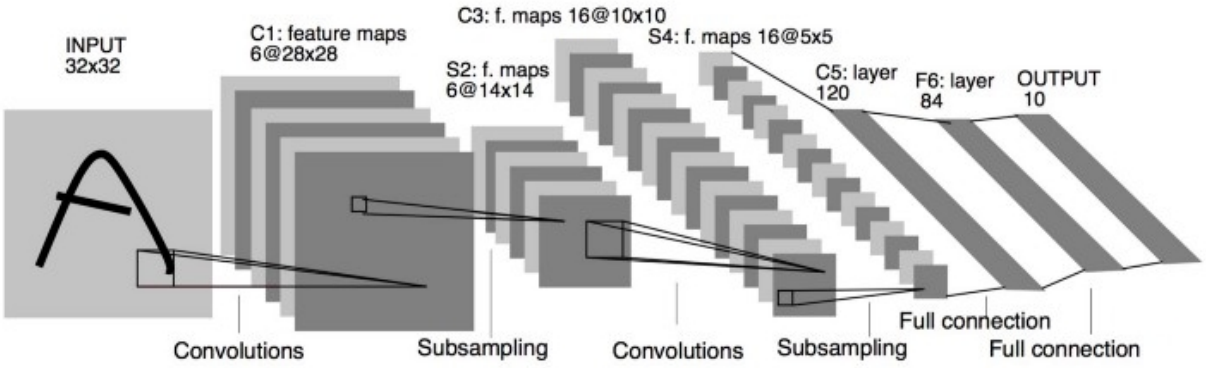

Figura 2.4: Arquitetura de uma CNN. (Lawrence et al., 1997)

permitem a identificação de padrões na imagem como linhas horizontais e verticais, curvas, texturas e formas.

A camada de subamostragem aplica uma redução na dimensionalidade dos blocos extraídos de uma camada de convolução.

As camadas totalmente conectadas são as mesmas camadas de uma rede neural artificial e são elas que fazem a classificação da imagem baseada nas informações extraídas pelas outras camadas.

2.6 CLASSIFICAÇÃO DOS ALGORITMOS DE APRENDIZAGEM POR REFORÇO

Os algoritmos de Aprendizagem por Reforço que utilizam redes neurais para realizar o treinamento dos agentes são classificados como algoritmos de Aprendizagem por Reforço Profundo, em inglês *Deep Reinforcement Learning* (DRL). Existem diversas abordagens para resolver problemas com DRL, como as baseadas em valor, modelo ou otimização de políticas. A Figura 2.5 ilustra a taxonomia dos algoritmos de DRL.

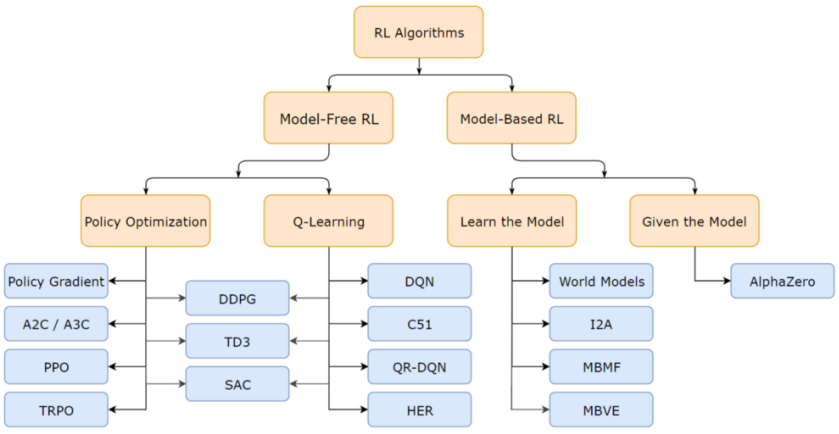

Figura 2.5: Classificação dos algoritmos de DRL. A principal classificação dos algoritmos é feita a partir uso (ou aprendizado) de um modelo do ambiente ou não. Dos algoritmos livres de modelo, é possível classificá-los em algoritmos baseados em política ou valor (*Q-Learning*). (OpenAI, 2018b)

2.6.1 Abordagens baseadas em modelo

Em aprendizagem por reforço, um **modelo** é um mapeamento de pares $(Ações, Estados)$ para distribuições probabilísticas sobre estados feito a partir das experiências do agente ou

previamente conhecido. Em outras palavras, o modelo é uma representação interna do agente sobre o ambiente que armazena as transições entre estados e as consequências das suas ações no ambiente. Os algoritmos que possuem um modelo do ambiente são classificados como algoritmos baseados em modelo.

Com o conhecimento sobre a estrutura e operação interna do ambiente, é possível ao agente determinar os próximos estados e recompensas que resultarão de suas ações e fazer um planejamento escolhendo as ações que darão a maior quantidade de recompensas, como ilustrado na Figura 2.6.

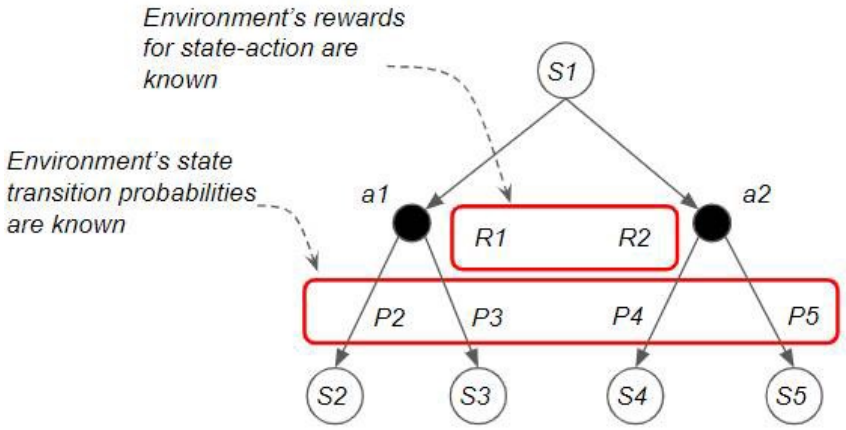

Figura 2.6: Em uma abordagem baseada em modelo, um agente consegue fazer o planejamento de suas ações que dão o melhor resultado por possuir informações sobre os estados e recompensas resultantes. (Doshi, 2020)

Segundo os autores Nagabandi et al. (2018), as abordagens baseadas em modelo possuem uma maior eficiência de amostragem que as abordagens livres de modelo, ou seja, precisam de menos exemplos de dados para operarem com sucesso no ambiente, já que elas podem apenas simular transições usando o modelo aprendido. Por outro lado, como esta classe de algoritmos necessita de um modelo global do ambiente, ambientes complexos são caros computacionalmente graças à maldição da dimensionalidade.

2.6.2 Abordagens livres de modelo

As abordagens livres de modelo não possuem ou não aprendem um modelo do ambiente como nas abordagens baseadas em modelo, já que o MDP é desconhecido. Em vez disso, o agente aprende usando apenas as suas experiências, explorando os estados via tentativa e erro, em busca de uma política ótima.

Esta classe de algoritmos pode ser dividida em duas categorias principais: baseadas em valor e baseadas na política.

Nas abordagens baseadas em política, o algoritmo otimiza a política diretamente, construindo uma representação dela e guardando-a em memória durante o treinamento. Um exemplo de algoritmo baseado em política é o *Proximal Policy Optimization* (Schulman et al., 2017).

Já nas abordagens baseadas em valor, não são armazenadas representações da política, apenas a função valor. O algoritmo tem como objetivo otimizar a função valor que é utilizada pelo agente ao escolher a ação que resultará no estado com mais recompensas. O *Deep Q-Networks* (Mnih et al., 2013) é um algoritmo que pertence a esta classe.

Também existem as abordagens híbridas que combinam as duas abordagens, como os algoritmos *Advantage Actor-Critic* (Wu et al., 2017) e *Sample Efficient Actor-Critic with Experience Replay* (Wang et al., 2016a).

2.7 DEEP Q-LEARNING

2.7.1 Q-Learning

Q-Learning é um algoritmo iterativo de RL baseado em valor e sem modelo utilizado para encontrar valores ação-estado ótimos $Q^*_\pi(s, a)$ em um MDP.

Os valores ação-estado $Q(s, a)$ são armazenados em uma tabela contendo os estados no eixo x e as ações no eixo y. A cada iteração do algoritmo, os valores da tabela são atualizados conforme a equação

$$Q_{t+1}(s_t, a_t) = (1 - \alpha)Q_t(s_t, a_t) + \alpha(r_t + \gamma \max(Q_t(s_{t+1}, a_t))), \tag{2.12}$$

onde γ é o fator de desconto, r_t é a recompensa recebida ao tomar a ação a_t no estado s_t e $\alpha \in (0, 1)$ é o fator de aprendizagem utilizado para limitar o passo de aprendizagem, fazendo uma média ponderada entre a informação nova e a informação antiga (Sutton e Barto, 2018). Utiliza-se a expressão $\max(Q_t(s_{t+1}, a_t)$ para obter o valor do próximo estado, comparando as estimativas de recompensas futuras e escolhendo a ação que resultará no estado com mais recompensas.

No início do algoritmo, a tabela não possui informações sobre os estados e recompensas e é inicializada com valores arbitrários. Por isso, é necessário explorar o ambiente passando por um mesmo estado múltiplas vezes e experimentando diferentes ações para popular os dados da tabela. Após um número suficiente de iterações, a função Q converge à uma política ótima (Sutton e Barto, 2018). A estratégia ϵ-*greedy* é bastante utilizada para priorizar a exploração do ambiente nas iterações inicias.

Q-Learning não é apropriado para resolver problemas com alta dimensionalidade por armazenar os valores do aprendizado em uma tabela. Ambientes com espaço de ações contínuo, por exemplo um robô que precisa controlar o ângulo de um braço, também dificultam o uso deste algoritmo. Uma possível solução é utilizar aproximação de funções como uma rede neural que é capaz de generalizar experiências passadas para uso em estados inéditos para o agente.

2.7.2 Deep Q-Learning

Proposto por Mnih et al. (2013), *Deep Q-Networks* (DQN) é um algoritmo iterativo de DRL baseado no algoritmo *Q-Learning* porém com a adição de redes neurais.

As redes neurais possibilitam a resolução de problemas com dimensões altas já que com elas não é necessário armazenar cada par estado-ação em memória nem revisitar cada estado individualmente, bastando apenas ajustar os valores dos pesos entre os neurônios.

É possível utilizar redes neurais de duas maneiras distintas nos problemas de DRL. Uma abordagem é usar um estado s e uma ação a como entrada de uma rede neural para estimar o valor de $Q(s, a)$. Outra abordagem é usar apenas o estado s como entrada para a rede e estimar um $Q(s, a)$ para cada ação a possível no problema. As duas abordagens são ilustradas na Figura 2.7.

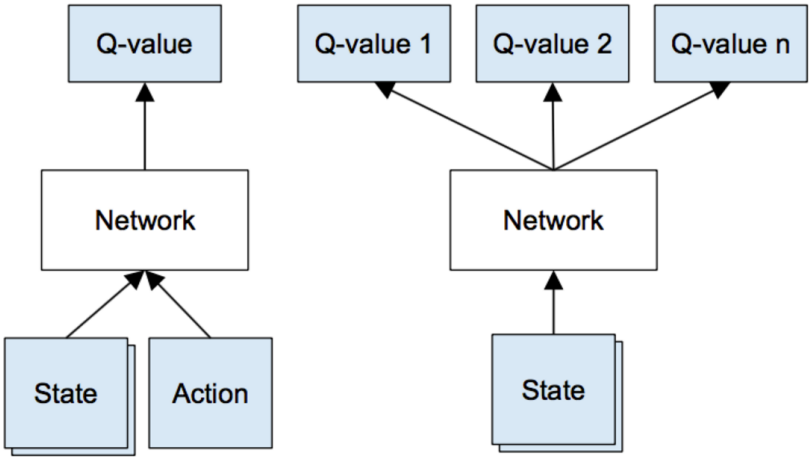

Figura 2.7: Possíveis tipos de uso de redes neurais em Deep Q-Learning. A rede pode receber como entrada uma observação do estado e uma ação e ter um Q-valor como saída, ou alternativamente, pode receber apenas a observação do estado como entrada e ter múltiplos Q-valores associados a cada ação possível.

A rede neural utilizada no algoritmo DQN em jogos de Atari 2600 é uma CNN, que é o tipo de rede mais adequado para o reconhecimento de padrões em imagens (LeCun et al., 1998). Para reduzir o custo computacional do algoritmo, é aplicado um pré-processamento na imagem de entrada, transformando-a em uma imagem em tons de cinza e redimensionando-a para uma imagem menor. Além disso, são utilizadas múltiplas imagens pré-processadas em frames consecutivos para representar uma única entrada. Isto é útil para capturar informações de direção e movimento de objetos em uma cena, como a direção que a bola está se movendo no jogo Pong em um dado momento. A Figura 2.8 ilustra uma arquitetura de uma CNN utilizada no algoritmo DQN.

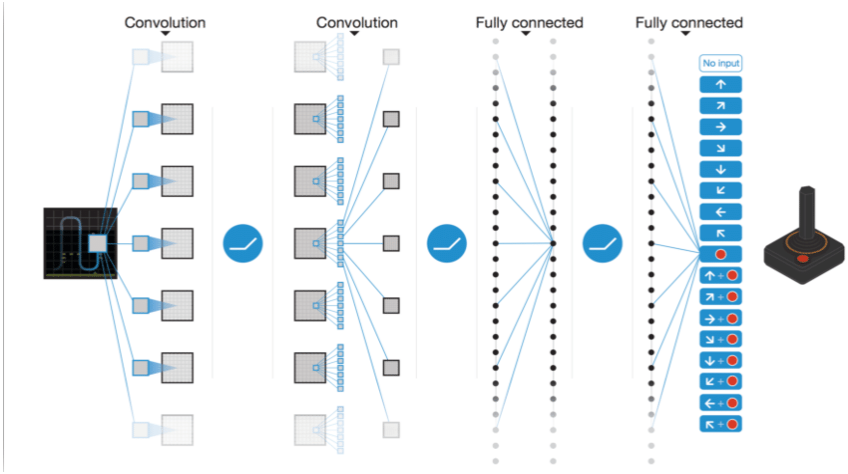

Figura 2.8: Exemplo de uso de uma CNN em um problema de DRL em jogos de Atari 2600. (Mnih et al., 2015)

No processo de aprendizagem de um algoritmo DRL, os *frames* ocorrem em sequência e com poucas alterações entre si e portanto dizemos que existe uma forte correlação entre as amostras, o que ocasiona em um *overfitting* na rede neural (Fenjiro e Benbrahim, 2018). O DQN aplica uma estratégica chamada *experience replay* para resolver o problema da correlação, em que é feito o armazenamento de uma quantidade de experiências do agente, dada pela tupla $(s_t, a_t, r_{t+1}, s_{t+1})$ que resume a experiência do agente em um tempo t. O aprendizado é então

realizado a partir de um conjunto de amostras aleatórias desta memória de experiências, o que quebra a correlação entre os dados (Fenjiro e Benbrahim, 2018).

A Equação 2.13 é utilizada como função de erro para atualizar os pesos da rede neural, onde $r_{t+1} + \gamma \max_a(Q(s_{t+1}, a))$ é chamado de alvo e $Q(s_t, a_t)$ é chamado de predição (Fenjiro e Benbrahim, 2018).

$$E = \frac{1}{2}[r_{t+1} + \gamma \max_a(Q(s_{t+1}, a)) - Q(s_t, a_t)]^2 \tag{2.13}$$

Observe que para esta equação é necessário realizar duas passagens pela rede neural utilizando os mesmos pesos, uma para calcular o $Q(s_t, a)$ do alvo e outra para o $Q(s_{t+1}, a)$ da predição. Ao otimizar os pesos da rede, estamos tentando aproximar a predição da rede para um alvo que está em movimento, o que causa uma instabilidade no treinamento.

Para resolver este problema, o DQN utiliza a técnica de alvo fixo que consiste em utilizar duas redes neurais, uma principal para o cálculo da predição e uma secundária para calcular o alvo. A rede alvo é uma cópia contendo os mesmos pesos da rede principal, seus pesos permanecem inalterados e só são atualizados a cada N iterações de treinamento. Desta forma, para realizar o cálculo da Equação 2.13 que atualiza a rede principal, utilizamos a rede principal para calcular a predição e a rede secundária para calcular o alvo.

Uma versão do algoritmo chamada *Dueling Deep Q-Networks* proposta por Wang et al. (2016b) divide o cálculo dos Q-valores em duas partes, uma calcula a função valor $V(s)$ e outra calcula uma função de vantagem $A(s, a)$, como na Equação 2.14. A Figura 2.9 mostra a diferença entre as arquiteturas DQN e *Dueling* DQN.

$$Q(s, a) = A(s, a) + V(s) \tag{2.14}$$

A vantagem nos diz quão melhor é escolher uma ação em comparação com as outras em termos de otimização, ou seja, quão longe a estimação do valor ação-estado está do valor do estado. Isto torna a estimação do valor de cada estado mais robusta, já que ele fica desassociado de uma ação específica, o que é especialmente útil em estados que em que as ações não afetam o ambiente significativamente (Wang et al., 2016b).

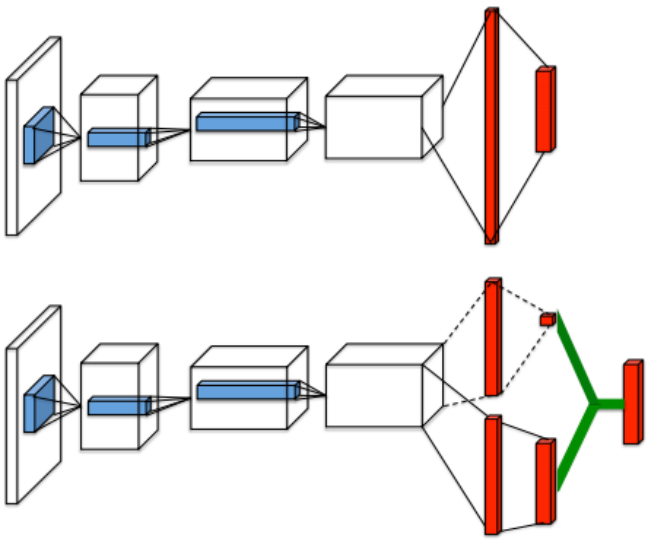

Figura 2.9: Arquiteturas dos algoritmos DQN (acima) e *Dueling* DQN (abaixo). (Wang et al., 2016b)

2.8 ADVANTAGE ACTOR CRITIC

Na seção anterior vimos que no algoritmo DQN, o agente aprende uma política ótima indiretamente a partir dos Q-valores em cada estado, que são então usados para derivar uma política.

Já nos algoritmos baseados em política estimamos a política diretamente sem usar uma função de valor, recebendo estados como entrada e gerando como saída uma distribuição de probabilidades para as ações.

Um algoritmo Ator-Crítico combina as abordagens baseadas em valor com as abordagens baseadas em política utilizando duas redes neurais, sendo uma chamada de ator que escolhe uma ação do agente para o estado atual (fazendo o papel da política) e outra chamada crítico que mensura a qualidade das ações tomadas pelo ator (fazendo o papel da função valor). As duas redes são atualizadas conforme o erro da estimativa do crítico (Sutton e Barto, 2018). A Figura 2.10 ilustra o funcionamento de um algoritmo Ator-Crítico.

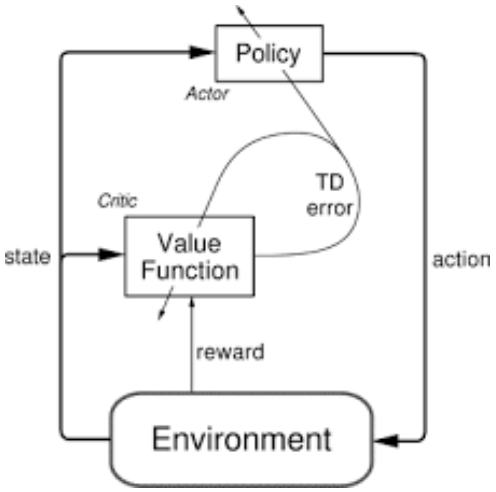

Figura 2.10: Arquitetura de um algoritmo Ator-Crítico. (Sutton e Barto, 2018)

Em outras palavras, o ator é uma função de política $\pi(s, a, \theta)$ enquanto o crítico calcula $q(s, a, w)$, sendo θ e w os parâmetros de suas respectivas redes neurais. Ambos são executados paralelamente e são aprimorados separadamente.

A cada iteração do algoritmo, um estado s_t é passado como entrada para o ator e para o crítico. O ator realiza a ação a_t dada pela sua política e recebe uma recompensa, enquanto o crítico calcula o valor de tomar a mesma ação naquele estado. A partir do valor $q(s, a)$ calculado pelo crítico, os pesos do ator são atualizados conforme a Equação 2.15 (Simioni, 2018).

$$\triangle\theta = \alpha\nabla_\theta(\log \pi_\theta(s_t, a_t)q(s_t, a_t)) \tag{2.15}$$

O ator então recebe o próximo estado s_{t+1} e realiza uma ação a_{t+1}. Com a nova ação, o crítico atualiza os seus parâmetros conforme a Equação 2.16 (Simioni, 2018). Observe que o ator e o crítico possuem taxas de aprendizagem diferentes (α e β).

$$\triangle w = \beta A(s_t, a_t)\nabla_w q(s_t, a_t) \tag{2.16}$$

Na equação anterior, A é a vantagem definida na Equação 2.17. A vantagem nos diz o quão melhor ou pior é uma ação em comparação ao valor previsto para o estado, o que permite o agente priorizar as ações em que a predição está errando mais.

$$A(s_t, a_t) = q(s_{t+1}, a_{t+1}) - V(s_t) \tag{2.17}$$

Os autores Mnih et al. (2016) construíram uma versão paralelizável deste algoritmo chamado *Asynchronous Advantage Actor-Critic* (A3C), em que vários atores são treinados ao mesmo tempo nas suas cópias do ambiente. Os atores são inicializados com pesos distintos, fazendo com que o ambiente seja explorado de diferentes maneiras. Além disso, há apenas um único crítico no algoritmo que recebe atualizações dos atores assincronamente e compartilha estas experiências com todos os outros atores.

Segundo os autores, a paralelização dos atores reduz o tempo de treinamento e também reduz a correlação entre os dados, já que os múltiplos atores estarão explorando estados diferentes em um dado momento.

Existe também a versão síncrona do mesmo algoritmo chamada *Advantage Actor-Critic (A2C)* (Wu et al., 2017), em que o crítico espera todos os atores terminarem o treinamento antes de atualizar os seus pesos para que os atores possam começar com a mesma política na próxima iteração. Isto faz com que o treinamento seja mais estável e potencialmente convergir mais rápido, segundo os autores. A diferença entre os dois algoritmos é ilustrada na Figura 2.11.

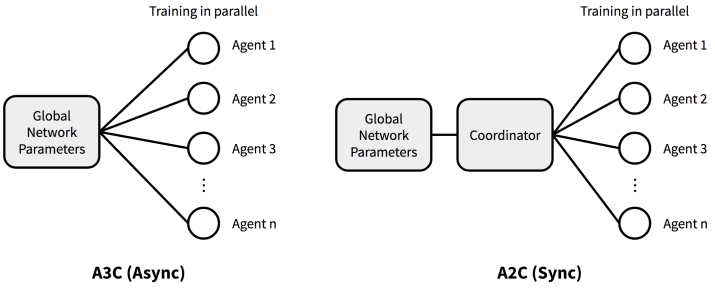

Figura 2.11: Diagrama dos algoritmos A3C e A2C com múltiplos atores e um crítico em comum. Os algoritmos diferenciam-se no sincronismo dos múltiplos atores em paralelo. (Sutton e Barto, 2018)

2.9 PROXIMAL POLICY OPTIMIZATION

Proximal Policy Optimization (PPO) é um algoritmo proposto por Schulman et al. (2017) baseado em ator-crítico de vantagem. O algoritmo é uma versão melhorada de outro algoritmo chamado *Trust Region Policy Optimization* (TRPO) (Schulman et al., 2015), que tem como ideia geral evitar atualizações de políticas muito acentuadas que podem causar instabilidades e erros na otimização do modelo.

Em algoritmos baseados em política, devemos otimizar uma política como na Equação 2.15 impulsionando o agente a tomar ações que gerem mais recompensas. O problema está no tamanho do passo da otimização, já que se o passo for pequeno corremos o risco de ter um treinamento muito lento, e se for grande demais corremos o risco de ter uma variabilidade alta, causando instabilidade no processo.

Em vez de utilizar $\log \pi$ para calcular o impacto das ações, como na Equação 2.15 do algoritmo A2C, o algoritmo utiliza a razão $rt(\theta)$ entre a probabilidade de tomar uma ação utilizando a política atual e a probabilidade de tomar a mesma ação utilizando a política anterior,

como na Equação 2.18. Quando a razão está entre 0 e 1 temos que a ação é menos provável para a política atual do que para a política antiga e o contrário ocorre quando a razão é maior que 1.

$$r_t(\theta) = \frac{\pi_{\theta_t}(a_t|s_t)}{\pi_{\theta_{t-1}}(a_t|s_t)} \tag{2.18}$$

Com isso, poderíamos otimizar a função de perda $L(\theta) = E[r_t(\theta)A_t]$. Entretanto, uma ação que é muito mais provável de ocorrer em uma política do que na política antecessora resultaria em uma atualização muito grande. Por este motivo, o algoritmo introduz limites de corte (*clipping*) sobre a razão de probabilidades, como na Equação 2.19, para que uma atualização não torne uma política extremamente diferente de sua versão anterior.

$$L^{clip}(\theta) = E[\min(r_t(\theta)A_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t)] \tag{2.19}$$

Na equação anterior, a função $clip$ limita a razão de probabilidades com um parâmetro de corte ϵ para que ela não seja menor que $1-\epsilon$ e nem maior que $1+\epsilon$. O resultado é o valor mínimo entre a função limitada e a função não limitada. O limite de corte é mostrado na Figura 2.12.

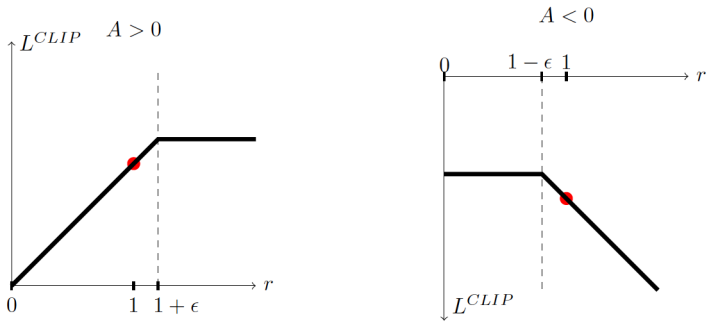

Figura 2.12: O limite de corte é mostrado para vantagens positivas à esquerda e vantagens negativas à direita. (Schulman et al., 2017)

Finalmente, a Equação 2.20 representa a função objetivo utilizada pelo algoritmo que utiliza a função $L^{clip}(\theta)$ definida anteriormente adicionada de um erro quadrático entre os dois valores e um fator de entropia S para o agente realizar explorações suficientes durante o treinamento, com os coeficientes c_1 e c_2 controlando o peso destes novos fatores.

$$L^{final}(\theta) = E[L^{clip}(\theta) - c_1(V_\theta(s) - V_{target})^2 + c_2 S[\pi_\theta](s_t)] \tag{2.20}$$

2.10 SAMPLE EFFICIENT ACTOR-CRITIC WITH EXPERIENCE REPLAY

O algoritmo *Sample Efficient Actor-Critic with Experience Replay* (ACER), proposto por Wang et al. (2016a) é um algoritmo ator-crítico que reúne diversas ideias implementadas em outros algoritmos DRL. Algumas destas ideias incluem o treinamento de vários atores em paralelo como no A3C, utilização de *experience replay* como no algoritmo DQN e otimização de política com região de confiança como no algoritmo PPO.

O algoritmo utiliza tanto uma abordagem baseada em política como uma abordagem baseada em valor. O agente obtém uma sequência de amostras e realiza um aprendizado baseado em política nela, em seguida armazena em um buffer para realizar outro aprendizado porém baseado em valor a partir de n amostras deste buffer. As atualizações do Q-valor têm como base outro algoritmo chamado Retrace(λ) (Munos et al., 2016).

A atualização incremental em Q^{ret} é utilizada tanto para a parte baseada em política como na parte baseada em valor do algoritmo, e o seu cálculo é realizado sobre uma trajetória de τ de experiências obtida do *buffer* de *experience replay*. A atualização é dada por

$$\Delta Q^{\text{ret}}(s_t, a_t) = \gamma_t \prod_{1 \leq \tau \leq t} \min(c, \omega_t)\delta_t, \tag{2.21}$$

onde ω_t é uma amostragem por importância utilizada para estimar a função valor para uma política π porém com amostras coletadas usando uma política de comportamento β utilizada para gerar o *buffer* de experiências, conforme a Equação 2.22 (Weng, 2018). Os produtos das amostragens podem causar altas variâncias e por isso utiliza-se um limite de corte por uma constante c.

$$\omega_t = \frac{\pi(A_\tau|S_\tau)}{\beta(A_\tau|S_\tau)} \tag{2.22}$$

A atualização da política na porção baseada em política do algoritmo é dada pela Equação 2.23.

$$\hat{g}_t^{\text{acer}} = \omega_t \big(Q^{\text{ret}}(s_t, a_t) - V_{\theta_v}(s_t)\big) \nabla_\theta \ln \pi_\theta(a_t|s_t) \tag{2.23}$$

O crítico é atualizado a partir da minimização do erro quadrático entre os Q-valores conforme

$$L = (Q^{\text{ret}}(s, a) - Q(s, a))^2 \tag{2.24}$$

2.11 TRANSFERÊNCIA DE APRENDIZADO

Transferência de Aprendizado é uma técnica de aprendizado de máquina em que um modelo treinado para realizar uma tarefa específica é utilizado para aprender uma segunda tarefa. A vantagem disto está na possibilidade de aprender uma nova tarefa aproveitando o conhecimento já adquirido anteriormente, dispensando a necessidade de treinar toda a rede novamente.

Redes pré-treinadas são frequentemente utilizadas como ponto de partida em diversos problemas de reconhecimento de imagens como a AlexNet (Krizhevsky et al., 2012), uma CNN disponibilizada publicamente e treinada com a enorme base de dados ImageNet (Deng et al., 2009), que possui cerca de 14 milhões de imagens divididas em 1000 classes e requer muito tempo e poder de processamento.

Este processo também é útil na área de DRL, onde o processo de aprendizagem livre de modelo tem uma eficiência baixa no uso de amostras de dados e é consequentemente lento (Shao et al., 2019). Além disso, é especialmente útil para agentes que atuam em um ambiente não-simulado onde provavelmente rodar milhões de iterações de aprendizado é impraticável (Arulkumaran et al., 2017).

Conforme (Glatt et al., 2016) onde são realizados experimentos de transferência de aprendizagem entre jogos de Atari 2600 utilizando o algoritmo DQN, foi verificado que o processo tende a funcionar melhor quando o jogo de origem do aprendizado é similar ao jogo alvo de aprendizado. Entretanto, quando os jogos de origem e alvo não são tão similares, o aprendizado é prejudicado, tendo uma piora na performance do agente quando em comparação com a inicialização aleatória dos pesos da rede (aprendizado sem transferência).

Quando sucedido, o processo de transferência de aprendizado gera uma melhora na performance da aprendizagem como mostra a Figura 2.13, onde idealmente se observam melhoras no início, na convergência e também na subida mais acentuada da curva de aprendizado.

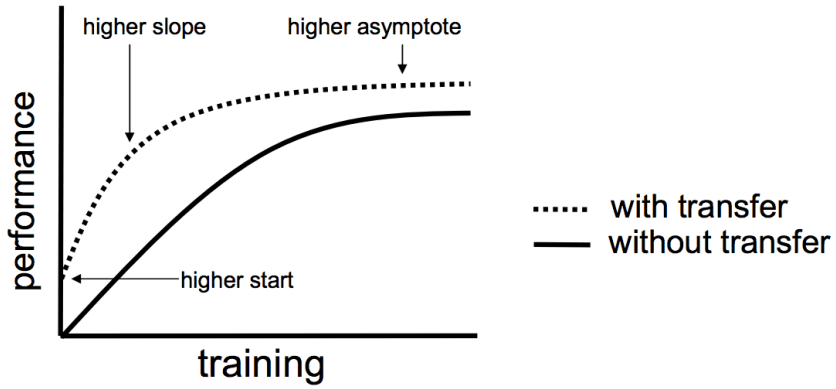

Figura 2.13: Três possíveis benefícios da transferência de aprendizado. (Olivas et al., 2009)

3 PROPOSTA

Este capítulo apresenta o sistema utilizado para o treinamento dos agentes de DRL em jogos de Atari 2600, descrevendo as características dos ambientes, ações, sistema de recompensas e os algoritmos utilizados.

3.1 FERRAMENTAS UTILIZADAS

A biblioteca *Arcade Learning Environment* (Bellemare et al., 2013) oferece um ambiente de emulação para jogos de Atari 2600, utilizando o emulador *Stella*, com funcionalidades que ajudam no desenvolvimento de agentes de inteligência artificial. Estas funcionalidades incluem a extração e padronização do sistema de pontuação entre os diferentes jogos da plataforma, emulação desacoplada dos módulos de renderização e som para obter melhor performance e ferramentas de visualização dos *frames* nos jogos.

OpenAI Gym (Brockman et al., 2016) é uma ferramenta de código aberto que reúne diversos ambientes para o desenvolvimento e comparação de algoritmos de RL, como jogos de Atari 2600 utilizando *Arcade Learning Environment* e também jogos de tabuleiro, ambientes de controle contínuo e simuladores 2D e 3D de robôs. Comparar algoritmos de RL é uma tarefa complexa, já que pequenas diferenças em seus parâmetros ou no sistema de recompensas podem alterar significativamente o resultado, dificultando também a reprodução de resultados. Por isso, *frameworks* como *OpenAI Gym* e *Arcade Learning Environment* são importantes na área de RL.

A biblioteca *OpenAI Baselines* (Dhariwal et al., 2017) é uma biblioteca de código aberto que possui implementações de alta qualidade de diversos algoritmos de DRL, fornecendo bases sólidas para pesquisadores da área. O *Tensorflow* (Abadi et al., 2015) é utilizado como base para treinamento e inferência das redes neurais utilizadas nos algoritmos de DRL. A biblioteca suporta os ambientes de *OpenAI Gym* mas também há a possibilidade de utilizar ambientes customizados.

Entretanto, como a biblioteca entrou em fase de manutenção e parou de ser atualizada, optou-se por utilizar o *fork* feito pela comunidade científica chamado *Stable Baselines* (Hill et al., 2018), que continuou os trabalhos realizados pela OpenAI adicionando documentação, reestruturando o código e adicionando mais algoritmos.

3.2 SISTEMA DE APRENDIZADO

3.2.1 Funcionamento Geral

A Figura 3.1 mostra o funcionamento do sistema proposto e as interações entre as ferramentas utilizadas, sendo que o *OpenAI Gym* fornece o ambiente e o Stable Baselines fornece o agente e o algoritmo de aprendizagem.

Cada iteração do sistema de aprendizagem consiste no agente observar um estado, predizer uma ação baseado na sua política, receber uma recompensa do ambiente e atualizar os parâmetros da política.

Um episódio é a execução de várias sequências de iterações de estados, ações e recompensas até o estado final do jogo. No *OpenAI Gym*, o ambiente dá um sinal de fim de jogo quando o jogador ganha uma partida, perde uma vida ou completa uma fase, sem a necessidade de perder todas as vidas até o *game over* ou completar o jogo, por exemplo. Além disso, os

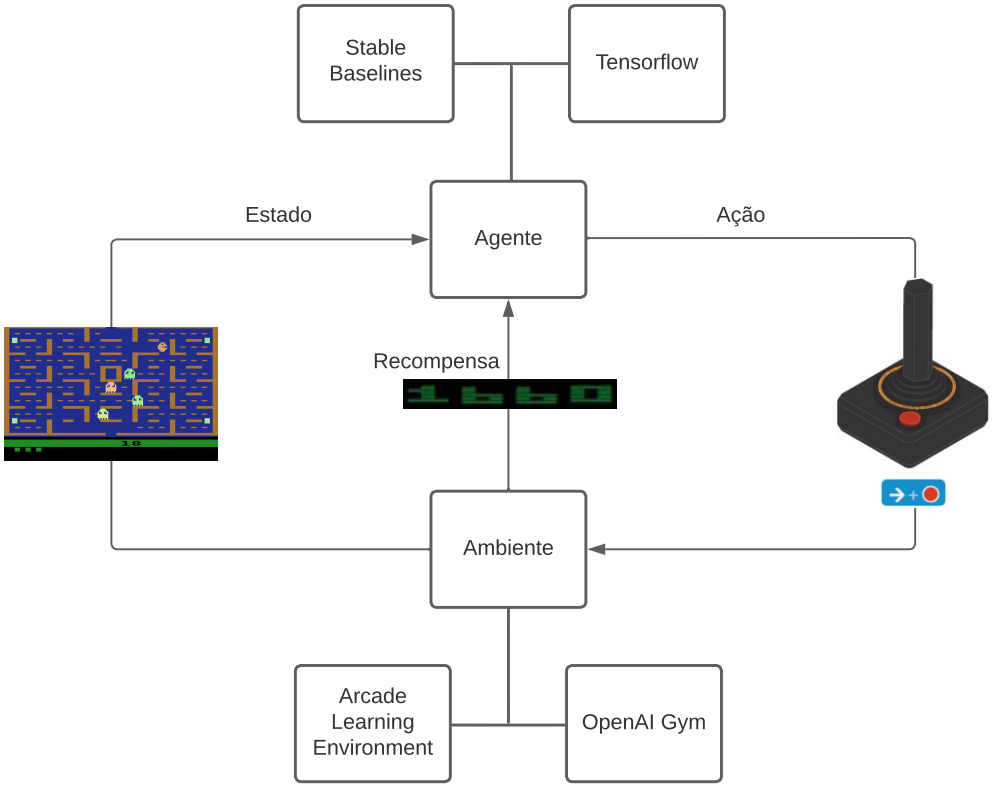

Figura 3.1: Funcionamento do sistema e interação entre as ferramentas utilizadas.

episódios têm um tempo máximo de execução (em tempo de jogo) para os casos em que o agente fica estagnado. No término de um episódio, o ambiente é reinicializado antes de começar o próximo episódio.

O processo de treinamento do agente é executado durante um número definido de iterações, no caso de jogos de Atari 2600 geralmente treinam-se os agentes por 10M iterações.

3.2.2 Ambiente

Foram selecionados 6 dos 59 jogos disponíveis no *OpenAI Gym* como ambientes neste trabalho, considerando diferentes gêneros e tamanho do espaço de ações.

O ambiente fornece ao agente uma imagem para cada *frame* no jogo, sendo representada por uma matriz de pixels. É realizado um pré-processamento na imagem para reduzir o custo computacional, reduzindo-a de uma imagem colorida de tamanho 160x210x3 para uma imagem em tons de cinza de tamanho 84x84. Cada pixel da imagem contém um valor entre 0 e 255 representando a intensidade da cor branca.

Em sequência, é aplicado o processo de *frame skipping* nos estados entregues pelo ambiente, que consiste em processar apenas um *frame* a cada 4 *frames*. Este processo visa reduzir o custo computacional do aprendizado sem perda significativa de informação, já que o emulador produz 60 *frames* por segundo, resultando em 15 *frames* por segundo com *frame skipping*. A última ação realizada pelo agente é repetida durante os *frames* ignorados.

Além disso, é feito uma concatenação de 4 *frames* em sequência para que o agente consiga capturar informações de direção e movimento em um dado estado, resultando em uma observação de tamanho 84x84x4. Todo o pré-processamento da entrada é ilustrado na Figura 3.2.

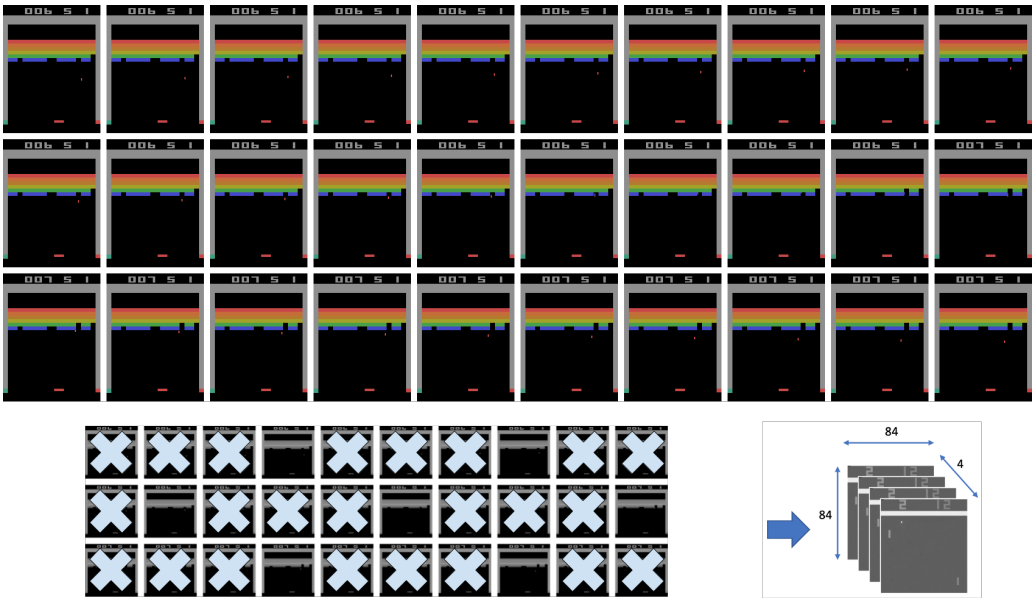

Figura 3.2: Entrada após a realização do pré-processamento, *frame skipping* de 4 *frames* e concatenação de 4 *frames* não ignorados consecutivos. Adaptado de (Seita, 2016) e (Torres, 2020).

3.2.3 Ações

O controle de um Atari 2600 possui um joystick de 8 direções e um botão. No *Arcade Learning Environment*, são definidos dois conjuntos discretos de ações: o conjunto completo de ações e o conjunto de ações úteis, contendo apenas as ações que são utilizadas em um determinado jogo.

O conjunto completo de ações inclui todas as 9 direções (Esquerda, Direita, Alto, Baixo, Esquerda-Alto, Esquerda-Baixo, Direita-Alto, Direita-Baixo e Vazio) permutando com as 2 ações válidas do botão (Pressionado e Não Pressionado), resultando em 18 ações possíveis.

No *OpenAI Gym*, é utilizado o conjunto de ações úteis, cujo tamanho varia de 3 a 18 ações dependendo do jogo executado. O número de ações influencia na saída das redes neurais convolucionais, que devem possuir o mesmo tamanho.

Alguns jogos como Pong e Breakout precisam de uma ação inicial pressionando o botão para que o jogo inicie, fazendo a bola se movimentar. Por este motivo, o *OpenAI Gym* executa a primeira ação automaticamente nos primeiros *frames* de cada episódio nestes jogos.

3.2.4 Recompensas

As recompensas são obtidas a partir da diferença entre as pontuações de um estado e o estado anterior, sendo $r_t = \text{score}_t - \text{score}_{t-1}$.

Como cada jogo possui um sistema de pontuação próprio, há uma grande variância entre cada jogo no Atari 2600. Por exemplo, a escala de pontuação no Pong é $[-21, 21]$, enquanto no Chopper Command a escala é $[0, 999900]$, o que faria a média de pontuação do segundo jogo dominar em uma comparação de performance com outros jogos.

Desta forma, o sistema de recompensas passa por um processo de padronização onde o agente passa a receber um valor entre $\{1, 0, -1\}$ como recompensa, sendo 1 para qualquer pontuações positivas, -1 para pontuações negativas e 0 quando a pontuação é inalterada.

Esta técnica limita a escala de recompensas e permite utilizar o mesmo fator de aprendizagem em múltiplos jogos, porém com a desvantagem de não ser possível para o agente diferenciar recompensas de diferentes magnitudes.

3.2.5 Avaliação

A performance de um agente durante o seu treinamento é medida a partir da sua própria pontuação obtida no jogo. Entretanto, como as recompensas são padronizadas e o funcionamento do jogo é alterado durante os episódios de treinamento, os agentes precisam ser avaliados em sessões separadas dos episódios de treinamento, sendo chamados de episódios de avaliação. Nestes, o agente não realiza nenhum aprendizado e é apenas avaliado conforme as pontuações sem padronização obtidas no jogo original, isto é, sem alterações no funcionamento do jogo como términos antecipados ao perder uma vida.

Os episódios de avaliação são executados 5 vezes a cada 10000 iterações de treinamento de um agente, sendo possível construir um gráfico da curva de aprendizagem, contendo os valores das pontuações finais a cada episódio de avaliação. Na literatura, a pontuação é geralmente comparada com as pontuações médias de humanos e agentes aleatórios para analisar o nível de aprendizagem obtido.

3.2.6 Algoritmos

A biblioteca *Stable Baselines* implementa diversos algoritmos de DRL, como mostrado na Tabela 3.1 que descreve o tipo de ação possível (discreta ou contínua) e o suporte a multiprocessamento para cada um deles.

Tabela 3.1: Algoritmos disponíveis na biblioteca Stable Baselines e suas características (Hill et al., 2018).

Algoritmo	Ações Contínuas	Ações Discretas	Multi processamento
A2C	✓	✓	✓
ACER	×	✓	✓
ACKTR	✓	✓	✓
DDPG	✓	×	✓
DQN	×	✓	×
HER	✓	✓	×
GAIL	✓	✓	✓
PPO1	✓	✓	✓
PPO2	✓	✓	✓
SAC	✓	×	×
TD3	✓	×	×
TRPO	✓	✓	✓

Como nos ambientes de jogos de Atari 2600 as ações são discretas, não seria possível utilizar os algoritmos DDPG, SAC e TD3 por suportarem apenas ações contínuas.

Em experimentos preliminares, o algoritmo ACKTR demandou um tempo de processamento significativamente maior que os demais (cerca de 10 horas de diferença) e por este motivo não foi utilizado.

Da classe de algoritmos de otimização de política com região de confiança, estão disponíveis os algoritmos TRPO, PPO1 e PPO2. Aqui, a biblioteca diferencia PPO2 do PPO1 com base na implementação: os algoritmos PPO1 e TRPO fazem o multiprocessamento apenas

pela CPU utilizando *OpenMPI* enquanto o PPO2 faz o processamento em GPU em múltiplos processos. Assim, foi escolhido apenas o PPO2 para este trabalho, já que ele é significativamente mais rápido no treinamento e é da mesma classe dos demais.

Já os algoritmos GAIL e HER fogem do escopo do trabalho. *General Adversarial Imitation Learning* (GAIL) é um algoritmo apropriado para aprendizagem por imitação, utilizando demonstrações de um humano, por exemplo. *Hindsight Experience Replay* (HER) é um método que roda em conjunto com outro algoritmo baseado em valor, sendo compatível apenas com DQN, SAC, TD3 e DDPG, e é utilizado para superar o problema de recompensas extremamente esparsas primariamente em problemas de robótica.

Assim, foram escolhidos os algoritmos DQN, A2C, ACER e PPO2 para a realização dos experimentos neste trabalho. Os parâmetros dos algoritmos e da rede neural convolucional utilizada encontram-se no Apêndice A.

4 RESULTADOS

Este capítulo apresenta os resultados obtidos a partir do treinamentos dos agentes utilizando os jogos de Atari 2600 escolhidos como ambiente. Além disso, são apresentados resultados de testes de Aprendizagem por Transferência entre dois jogos.

4.1 ROTEIRO DOS TESTES

Foram escolhidos os jogos *Pong*, *Ms. Pac-Man*, *Seaquest* e *Montezuma's Revenge* para serem utilizados como ambientes de teste e comparação dos agentes treinados com os algoritmos A2C, ACER, DQN e PPO2. Cada jogo possui um gráfico que mostra a performance dos agentes a partir de suas pontuações obtidas durante os períodos de avaliação nas 10 milhões de iterações.[1] Como as pontuações apresentam alta variabilidade, todos os gráficos apresentam os dados utilizando uma média móvel de 100 períodos. Para efeitos de comparação, os gráficos também possuem as pontuações de um agente aleatório (que sempre escolhe ações aleatórias) e as pontuações médias de humanos com base em (Lazaridis et al., 2020).

Também são mostrados os resultados dos experimentos de Aprendizagem por Transferência realizados entre os jogos *Space Invaders* e *Demon Attack*. Inicialmente são treinados os agentes em um jogo e em sequência em outro, utilizando os mesmos agentes com o aprendizado do primeiro jogo como ponto de partida. Em seguida, é realizado o mesmo tipo de experimento porém trocando os jogos de origem e destino.

O computador utilizado para os testes segue as especificações da Tabela 4.1.

Tabela 4.1: Características técnicas do computador utilizado.

CPU	AMD Ryzen 5 3600 6-Cores 12-Threads com clock de 3.6 GHz (4.2 GHz Turbo) e 32 MiB de cache
GPU	NVIDIA Geforce GTX 1060 com GDDR5 de 6 GiB
RAM	16GB DDR4 3000 MHz
Armazenamento	SSD primário de 256 GiB e HD de 2 TiB
Sistema Operacional	Pop!_OS 21.04
Versões das bibliotecas	Conda 4.11.0 CUDA Toolkit 9.0 cuDNN 7.6.5 Tensorflow 1.15.0 OpenAI Gym 0.21.0 Arcade Learning Environment 0.7.3 Stable Baselines 2.10.2

[1] O comportamento dos agentes nestes jogos pode ser observado nesta *playlist* de vídeos: `https://www.youtube.com/playlist?list=PLX-_JZGspmoPxPMX7n-HVSjSkcWmhTeuz`

4.2 COMPARAÇÃO ENTRE OS ALGORITMOS

4.2.1 Pong

Pong é um jogo eletrônico de duas dimensões que simula partidas de tênis de mesa. O jogador controla uma raquete do lado direito da tela, enquanto uma inteligência artificial predefinida controla a outra raquete no lado esquerdo. O objetivo do jogo é rebater a bola até o outro lado da tela de forma que ela ultrapasse o alcance do oponente, fazendo um ponto. A bola gradualmente aumenta a sua velocidade a cada vez que é rebatida, reiniciando no começo de outro episódio. A partida acaba quando um dos jogadores consegue 21 pontos no total, sendo ele o vencedor. A Figura 4.1 mostra uma captura de tela do jogo.

O espaço de ações úteis disponível no ambiente consiste em 3 ações: mover para cima, mover para baixo e não realizar movimento.

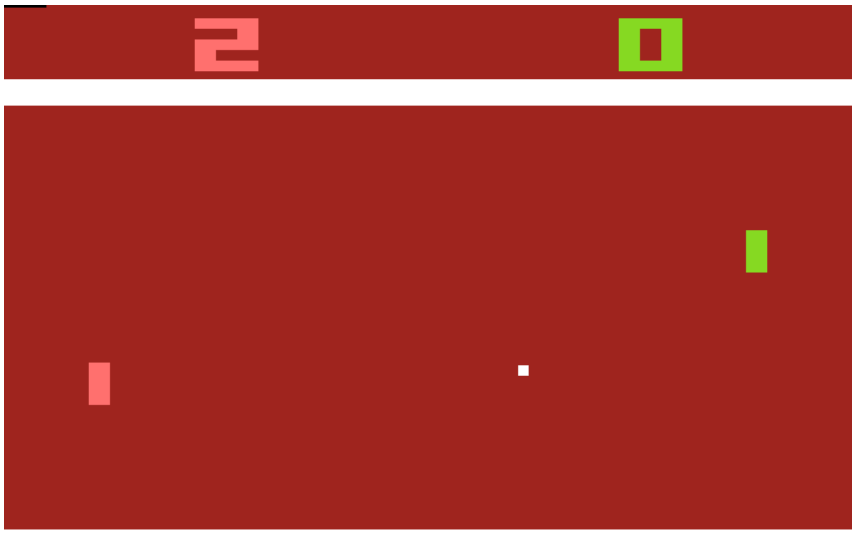

Figura 4.1: Captura de tela do jogo *Pong*.

Nos testes de avaliações dos agentes pelo ambiente provido pelo OpenAI Gym, as recompensas variam entre -21 e 21. O agente recebe uma recompensa positiva quando faz um ponto contra o adversário e recebe uma recompensa negativa quando falha em rebater a bola.

A evolução dos agentes neste ambiente está representada no gráfico da Figura 4.2, que mostra as pontuações obtidas por cada um dos agentes nos períodos de avaliação durante as 10 milhões de iterações de treinamento. O tempo total de cada treinamento é descrito na Tabela 4.2.

Tabela 4.2: Tempo total de treinamento para cada agente no jogo *Pong*.

A2C	ACER	DQN	PPO2
13h 27m	14h 49m	18h 17m	14h 4m

A partir do gráfico, é possível observar que os quatro agentes conseguiram realizar um aprendizado no ambiente, já que as suas pontuações foram melhores que a de um agente aleatório e até ultrapassam a de um humano.

O agente DQN convergiu a um ponto máximo mais rapidamente que os demais e não demonstrou variabilidade durante o resto do treinamento. O agente PPO2 inicialmente atingiu um patamar menor que o DQN porém subiu gradualmente até convergir a um ponto máximo, sem demonstrar variabilidade depois disso. Já o agente ACER demorou um pouco mais para

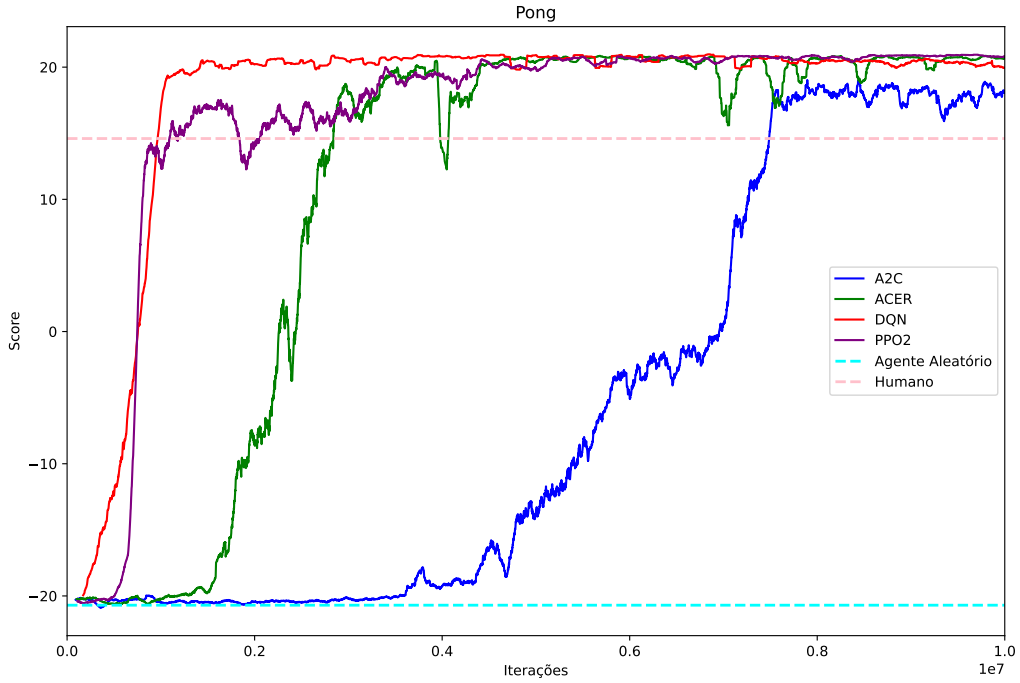

Figura 4.2: Pontuações obtidas pelos agentes nos episódios de avaliação durante o treinamento no jogo *Pong*.

convergir até um ponto máximo e além disso demonstrou uma certa variabilidade em alguns pontos depois de convergir. Enquanto isso, o A2C foi o agente que mais demorou até realizar algum aprendizado e convergiu em uma pontuação ligeiramente menor que os demais.

4.2.2 Ms. Pac-Man

Ms. Pac-Man é um jogo em duas dimensões onde o jogador deve navegar por um labirinto, coletando moedas ao mesmo tempo que foge de fantasmas que o perseguem. O jogador perde uma vida quando colide com um fantasma, e perde o jogo quando três vidas são perdidas. As *power pills*, distribuídas pelo labirinto e representadas por moedas maiores, têm o propósito de subverter o modo de operação do jogo por um breve momento. Ao serem coletadas, o jogador é capaz de capturar os fantasmas que passam a fugir ao invés de persegui-lo. A Figura 4.3 mostra uma captura de tela do jogo.

O espaço de ações úteis disponível no ambiente consiste em 9 ações de movimento, sendo 8 direções distintas e 1 ação sem movimento.

O jogo dá uma pontuação positiva quando o jogador coleta uma moeda (10 pontos), uma *power pill* (50 pontos) ou fantasmas (200 pontos) e não dá recompensas negativas, já que o placar não se altera com as mortes do jogador.

A evolução dos agentes neste ambiente está representada no gráfico da Figura 4.4, que mostra as pontuações obtidas por cada um dos agentes nos períodos de avaliação durante as 10 milhões de iterações de treinamento. O tempo total de cada treinamento é descrito na Tabela 4.3.

Figura 4.3: Captura de tela do jogo *Ms. Pac-Man*.

Figura 4.4: Pontuações obtidas pelos agentes nos episódios de avaliação durante o treinamento no jogo *Ms. Pac-Man*.

Tabela 4.3: Tempo total de treinamento para cada agente no jogo *Ms. Pac-Man*.

A2C	ACER	DQN	PPO2
4h 23m	6h 30m	11h 39m	5h 25m

A partir do gráfico, é possível observar que os quatro agentes conseguiram pontuações melhores que a de um agente aleatório, o que poderia significar que um conseguiram realizar um aprendizado. Entretanto, os agentes obtiveram uma pontuação em geral menor que a metade da pontuação de um humano, o que indica que provavelmente aprenderam a se locomover pelo

labirinto coletando moedas, porém não conseguiram efetivamente aprender toda a mecânica do jogo de fugir e capturar fantasmas.

O agente ACER obteve a melhor pontuação de todos os agentes, com pequenos saltos na pontuação logo antes do fim do treinamento. Enquanto isso, os outros agentes apresentaram uma curva de aprendizagem quase constante, com uma pequena ascensão na pontuação.

4.2.3 Seaquest

Seaquest é um jogo onde o jogador controla um submarino que deve atirar em inimigos e resgatar mergulhadores. O jogador possui munição ilimitada porém deve se atentar ao seu nível de oxigênio, que é consumido gradualmente e renovado ao retornar o submarino à superfície. O objetivo do jogador é resgatar os mergulhadores e levá-los até a superfície, sendo possível carregar até 6 mergulhadores de uma vez. O jogo aumenta a dificuldade a cada volta do jogador à superfície, e retira uma de suas vidas caso ele volte sem nenhum mergulhador. A Figura 4.5 mostra uma captura de tela do jogo.

Figura 4.5: Captura de tela do jogo *Seaquest*.

O espaço de ações úteis disponível no ambiente consiste em todas as 18 ações do espaço de ações. Todas as direções podem ser utilizadas em conjunto com o botão de atirar.

O jogador recebe uma pontuação 20 para cada tubarão ou submarino inimigo destruído e 50 para cada mergulhador resgatado, sendo que estas recompensas aumentam gradativamente a cada volta à superfície com mergulhadores resgatados.

A evolução dos agentes neste ambiente está representada no gráfico da Figura 4.6, que mostra as pontuações obtidas por cada um dos agentes nos períodos de avaliação durante as 10 milhões de iterações de treinamento. O tempo total de cada treinamento é descrito na Tabela 4.4.

Tabela 4.4: Tempo total de treinamento para cada agente no jogo *Seaquest*.

A2C	ACER	DQN	PPO2
5h 58m	6h 57m	14h 10m	5h 27m

A partir do gráfico, observa-se que houve um certo aprendizado, já que as pontuações foram melhores que a de um agente aleatório. Entretanto, as pontuações ficaram significativamente menores da média de pontuação humana 42054.7, o que sugere que os agentes não obtiveram conhecimento de toda a dinâmica do jogo.

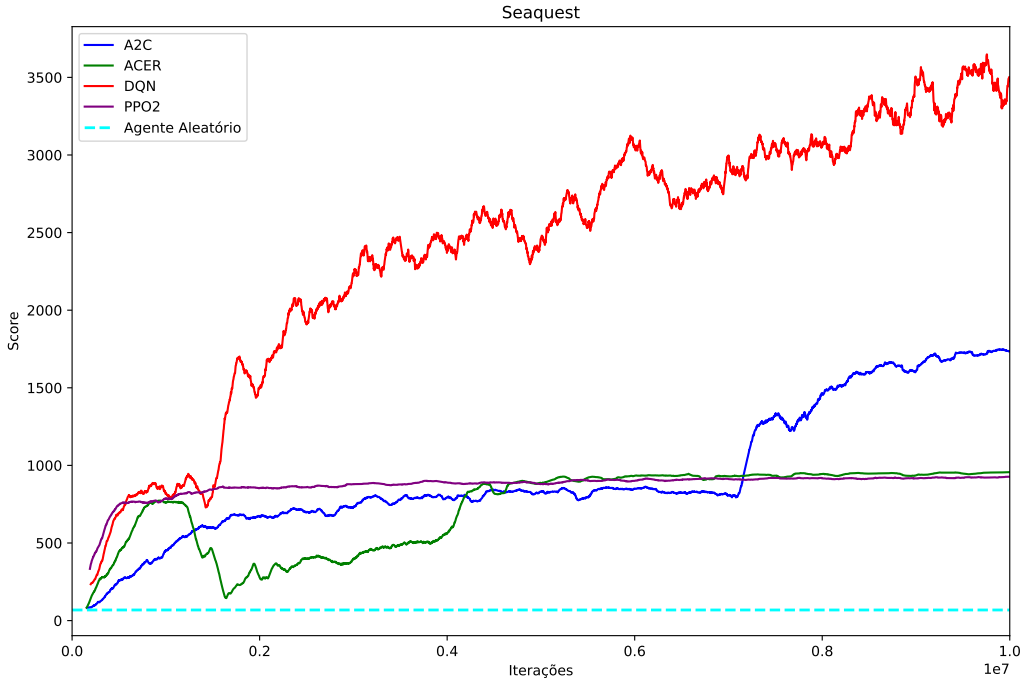

Figura 4.6: Pontuações obtidas pelos agentes nos episódios de avaliação durante o treinamento no jogo *Seaquest*. A média de pontuação humana para este jogo é de 42054.7, que foi omitida no gráfico por questões de escala.

Os agentes PPO2 e ACER convergiram em pontuações próximas, sendo que o PPO2 convergiu mais cedo. Inicialmente, o agente A2C convergiu em um ponto próximo dos agentes PPO2 e ACER, porém em um determinado momento convergiu em outra pontuação mais alta. Enquanto isso, o agente DQN ultrapassou o ponto de convergência dos outros agentes já nos momentos iniciais do treinamento e adquiriu a melhor performance, terminando com a curva de aprendizado ainda em ascensão.

4.2.4 Montezuma's Revenge

Montezuma's Revenge é um jogo de plataforma onde o jogador deve desviar de perigos como criaturas e abismos, procurar chaves que abrem portas e utilizar equipamentos encontrados ao longo do caminho como tochas para iluminar áreas escuras e espadas para atacar inimigos. O jogo contém múltiplas salas diferentes, sendo que pode ser necessário revisitá-las em certos momentos. A Figura 4.7 mostra uma captura de tela do jogo.

O espaço de ações úteis disponível no ambiente consiste em todas as 18 ações do espaço de ações. Todas as direções são utilizadas em conjunto com o botão de atacar.

O jogador recebe pontos para cada chave, espada, amuleto, joia e tocha encontrada, inimigo derrotado ou porta aberta (respectivamente, 50, 50, 100, 1000, 3000, 2000 e 300 pontos).

A evolução dos agentes neste ambiente está representada no gráfico da Figura 4.8, que mostra as pontuações obtidas por cada um dos agentes nos períodos de avaliação durante as iterações de treinamento. O tempo total de cada treinamento é descrito na Tabela 4.5.

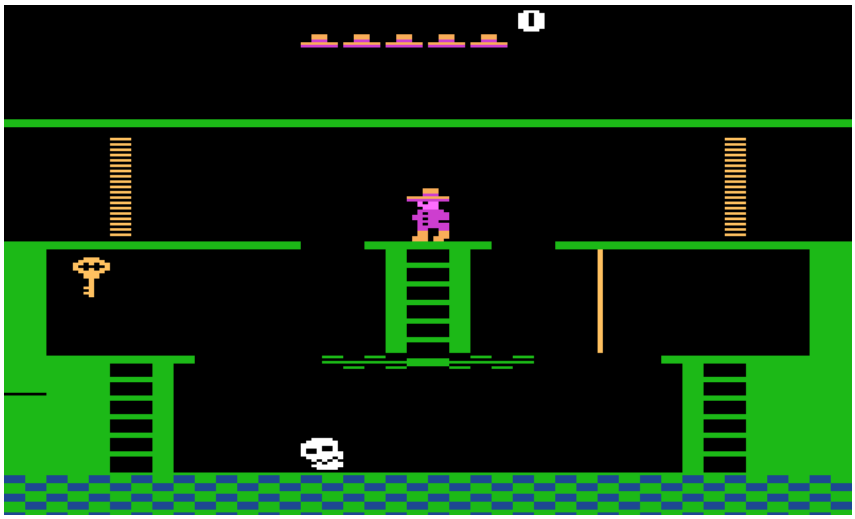

Figura 4.7: Captura de tela do jogo *Montezuma's Revenge*.

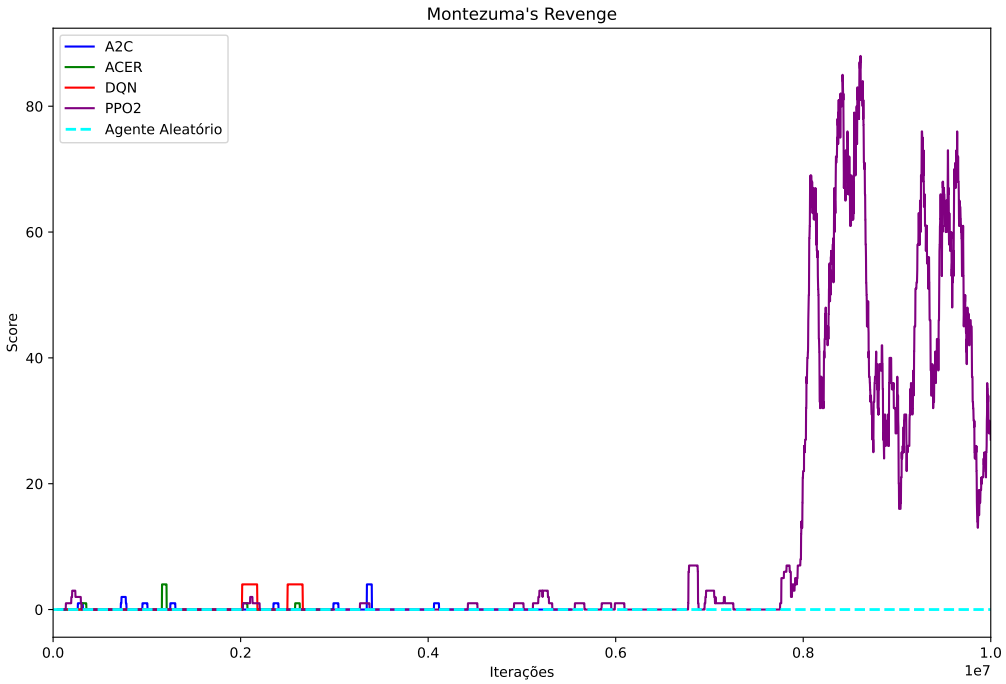

Figura 4.8: Pontuações obtidas pelos agentes nos episódios de avaliação durante o treinamento no jogo *Montezuma's Revenge*. A média de pontuação humana para este jogo é de 4753.3, que foi omitida no gráfico por questões de escala.

Tabela 4.5: Tempo total de treinamento para cada agente no jogo *Montezuma's Revenge*.

A2C	ACER	DQN	PPO2
2d 13h	2d 13h	2d 13h	1d 20h

O ambiente do jogo *Montezuma's Revenge* tem o problema de recompensas esparsas, ou seja, é necessário realizar uma longa sequência de ações até conseguir alguma recompensa.

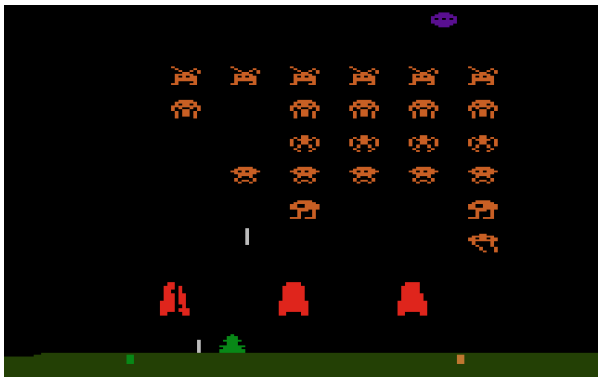

Figura 4.9: Captura de tela do jogo *Space Invaders*.

Figura 4.10: Captura de tela do jogo *Demon Attack*.

Desta forma, é difícil para o agente agente obter qualquer recompensa do ambiente executando apenas ações aleatórias e assim realizar algum aprendizado.

Como esperado, nenhum agente conseguiu obter uma performance melhor que o agente aleatório, com exceção do agente PPO2 que conseguiu algum resultado consistentemente diferente de zero. Mesmo assim, todos os agentes tiveram uma performance expressivamente pior que a média de pontuação humana 4753.3.

Observe que este experimento em especial não foi executado por 10 milhões de iterações como nos anteriores, já que os agentes A2C, ACER e DQN chegaram a 2 dias e 13 horas de execução totalizando, respectivamente, 2.52, 2.97 e 2.74 milhões de iterações. Já o agente PPO2 foi treinado por 10 milhões de iterações em 1 dia e 20 horas. O tempo elevado de treinamento é consequência do problema de recompensas esparsas, que faz com que os agentes não sejam estimulados a realizar nenhuma tarefa específica, aumentando consideravelmente o tamanho de cada episódio.

4.3 TRANSFERÊNCIA DE APRENDIZADO

4.3.1 Ambientes

No jogo *Space Invaders*, o jogador deve defender o seu território de um esquadrão de inimigos alienígenas que tentam invadir. Com este objetivo, o jogador controla uma espaçonave que é capaz de movimentar-se para os lados e atirar, além de contar com barreiras fixas no cenário que bloqueiam alguns projéteis vindos dos inimigos até serem destruídas. Quando todos os inimigos na tela são derrotados, um novo esquadrão de invasores aparece e o jogo aumenta de dificuldade. O ciclo se repete até o jogador perder todas as suas vidas, ao ser atingido ou deixar algum alienígena chegar à base.

Já o *Demon Attack* é um jogo semelhante ao *Space Invaders*, em que o jogador também controla uma espaçonave capaz de destruir inimigos. Entretanto, não existem barreiras protetoras no cenário e há uma quantidade menor de inimigos por vez na tela que se movem rapidamente e atiram com maior frequência, além de alguns se dividirem em dois inimigos menores quando derrotados, e também não há penalização para inimigos que chegam até a base do jogador. Quando todos os inimigos são derrotados, outros inimigos aparecem e o ciclo se repete, aumentando gradualmente de dificuldade, até o jogador perder todas as suas vidas sendo atingido por um ataque ou colidindo com um inimigo. As Figuras 4.9 e 4.10 mostram capturas de tela dos dois jogos.

O espaço de ações úteis de ambos os jogos incluem 6 ações distintas: mover-se para a esquerda ou direita ou não se mover, em conjunto com a ação de atirar ou não atirar.

No *Space Invaders*, o jogador ganha pontos para cada inimigo derrotado, sendo que os pontos aumentam dependendo da camada em que o inimigo se encontra no esquadrão (variando de 5 pontos para inimigos na primeira camada, 10 na segunda e assim sucessivamente até a última camada onde os inimigos valem 30 pontos). Já no *Demon Attack*, os inimigos começam valendo 10 pontos e aumentam gradualmente até a 11ª fase em diante, onde os inimigos simples passam a valer 35 e os que se dividem valem 70.

Os dois jogos apresentam várias semelhanças entre si, como o mesmo espaço de ações e o mesmo estilo de jogo, além de serem visualmente parecidos já que em ambos os jogos a nave controlada pelo jogador se movimenta horizontalmente na parte inferior da tela enquanto se defende de inimigos que vêm de cima.

Dadas as semelhanças entre os dois jogos, um experimento de transferência de aprendizado entre eles pode ser realizado para analisar algum aproveitamento de conhecimento ao utilizar o treinamento prévio realizado em um jogo para treinar o outro. Poderíamos supor que um agente que já aprendeu a atirar e desviar de ataques de inimigos no *Space Invaders* teria uma vantagem inicial sobre um agente começando com pesos aleatórios no treinamento do jogo *Demon Attack*, por exemplo.

4.3.2 Experimentos

4.3.2.1 Experimento 1

Em um primeiro experimento, para cada agente é realizado um treinamento utilizando o jogo *Space Invaders* e em seguida, os mesmos pesos do agente são utilizados para outro treinamento no jogo *Demon Attack*. Outro agente é treinado com o jogo *Demon Attack*, porém sem utilizar outro treinamento como ponto de partida. Os resultados das pontuações obtidas durante os períodos de avaliação podem ser visualizados na Figura 4.11.

Para o algoritmo A2C, pode-se observar que houve uma melhora significativa na performance do agente com aprendizado prévio no jogo *Space Invaders* em comparação com outro agente apenas com inicializações aleatórias, onde as formas das curvas de aprendizagem são semelhantes na inclinação de subida porém com distância de cerca de 2500 entre as pontuações.

Entretanto, para os algoritmos ACER e PPO2 não houve alteração significativa no desempenho entre os agentes com inicialização aleatória e com transferência de aprendizado.

Já no algoritmo DQN, observa-se uma queda no desempenho do agente com transferência de aprendizado em relação ao agente com inicialização aleatória, com diferenças nas partes iniciais e finais da curva de aprendizagem, o que pode indicar uma transferência destrutiva.

Em geral, os resultados obtidos neste experimento não foram satisfatórios, já que esperava-se que os agentes com aprendizado prévio em um jogo semelhante teriam uma performance significativamente melhor, principalmente no início do treinamento. Entretanto, isto só ocorreu para o agente A2C. Além disso, outros agentes tiveram até uma piora na performance, como visto no agente DQN.

4.3.2.2 Experimento 2

Em um segundo experimento, o inverso do primeiro experimento é realizado, onde para cada algoritmo treinam-se dois agentes no jogo *Space Invaders*, sendo que um conta com inicializações aleatórias e outro possui treinamento prévio no jogo *Demon Attack*. Os resultados das pontuações obtidas durante os períodos de avaliação podem ser visualizados na Figura 4.12.

Para o agente com algoritmo ACER, observa-se uma melhora na pontuação do agente com transferência de aprendizado do início até a metade do treinamento, porém em seguida a

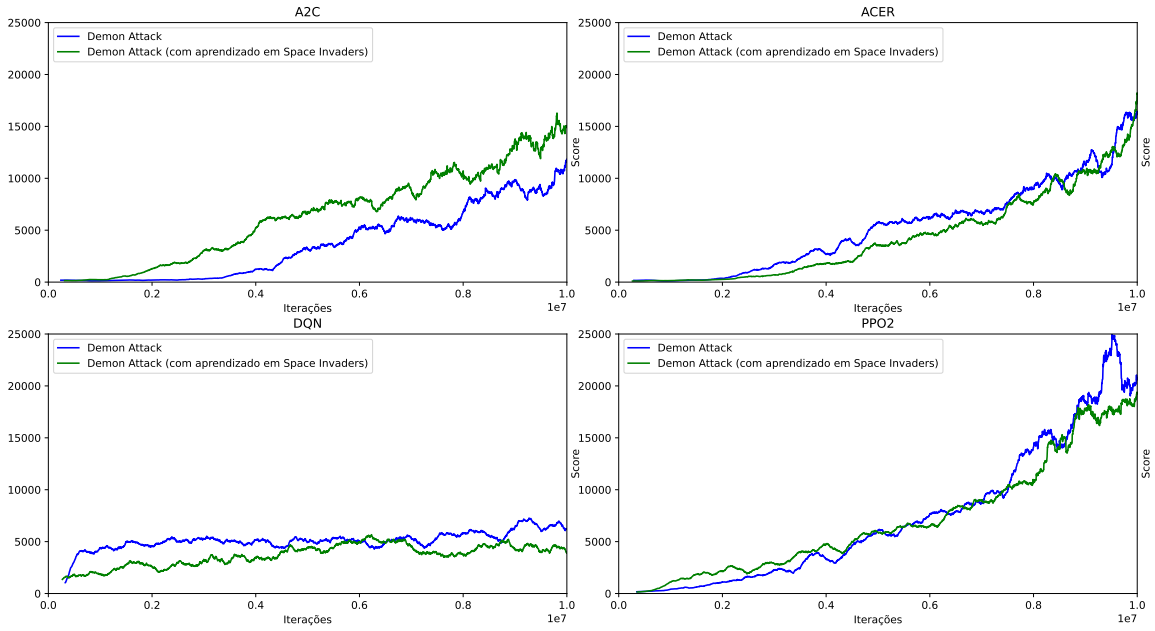

Figura 4.11: Pontuações obtidas pelos agentes nos episódios de avaliação durante o treinamento no jogo *Demon Attack* em comparação com as pontuações obtidas no mesmo jogo porém com treinamento prévio no jogo *Space Invaders*.

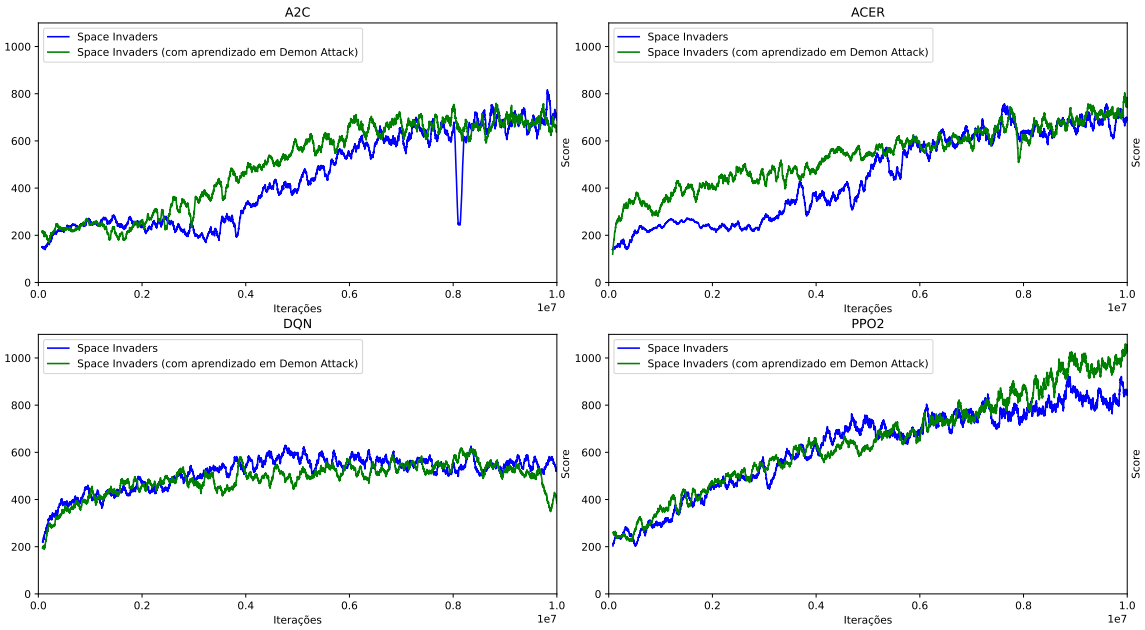

Figura 4.12: Pontuações obtidas pelos agentes nos episódios de avaliação durante o treinamento no jogo *Space Invaders* em comparação com as pontuações obtidas no mesmo jogo porém com treinamento prévio no jogo *Demon Attack*.

curva de aprendizagem acaba coincidindo aproximadamente nos mesmos valores do agente com inicialização aleatória.

No agente A2C observa-se também uma melhora na pontuação nos momentos perto da metade do processo de treinamento, porém as pontuações convergem em um mesmo ponto.

Já os agentes DQN e PPO2 não apresentam muitas diferenças nas curvas de aprendizagem, exceto por diferenças no fim dos treinamentos dos agentes que podem estar associadas a variabilidade natural de experimentos em aprendizagem por reforço e não necessariamente uma consequência da transferência de aprendizado.

Assim como no primeiro experimento, os resultados não apontaram uma diferença significativa no aprendizado por transferência entre um jogo e outro. Somente o agente ACER obteve uma melhora no início do treinamento, mas todos os agentes convergiram em valores semelhantes.

5 CONCLUSÃO

Este trabalho apresentou o desenvolvimento de agentes de IA utilizando quatro algoritmos distintos de DRL em ambientes de jogos eletrônicos do console Atari 2600. Os agentes utilizam redes neurais convolucionais para escolher as melhores ações a serem realizadas baseadas na imagem da tela dos jogos, treinando-as a partir de recompensas obtidas pela pontuação no jogo.

Foi realizado um experimento para analisar e comparar a performance dos quatro agentes em quatro jogos de características distintas. Considerando os resultados obtidos, observou-se que no jogo *Pong*, todos os agentes ultrapassaram a média de pontuações humanas. Nos jogos *Seaquest* e *Ms. Pac-Man*, houve um certo aprendizado porém com pontuação significativamente aquém do que humanos são capazes de obter. E no jogo *Montezuma's Revenge*, não houve aprendizado em nenhum dos agentes graças ao problema de recompensas esparsas presente no jogo. Além disso, nenhum dos agentes foi consistente melhor que outro nestes jogos, o que indica a importância do algoritmo utilizado para cada tipo de ambiente.

Também foram realizados experimentos com transferência de aprendizado entre dois jogos semelhantes (*Space Invaders* e *Demon Attack*) para analisar um possível aproveitamento de conhecimento, comparando as pontuações de agentes com transferência de aprendizado com as pontuações de agentes com inicialização de pesos aleatória. Considerando os resultados obtidos, não foi possível verificar diferenças significativas no desempenho dos agentes ao utilizar esta técnica.

Finalmente, propõem-se os seguintes trabalhos futuros:

- Alterar as redes neurais utilizadas nos agentes para uma combinação da rede neural convolucional com uma rede LSTM (*Long Short-Term Memory*), que pode ser útil em ambientes parcialmente observáveis pelo agente.

- Utilizar outros ambientes para testes, em especial o ViZDoom que oferece uma interface amigável a pesquisas para o jogo Doom, sendo altamente flexível na customização de cenários, no sistema de recompensas e até no funcionamento do jogo em si.

REFERÊNCIAS

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y. e Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Arulkumaran, K., Deisenroth, M. P., Brundage, M. e Bharath, A. A. (2017). A brief survey of deep reinforcement learning. *arXiv preprint arXiv:1708.05866*.

Bellemare, M. G., Naddaf, Y., Veness, J. e Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J. e Zaremba, W. (2016). Openai gym.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K. e Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. Em *2009 IEEE conference on computer vision and pattern recognition*, páginas 248–255. Ieee.

Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S., Wu, Y. e Zhokhov, P. (2017). Openai baselines. `https://github.com/openai/baselines`.

Doshi, K. (2020). Reinforcement learning made simple (part 2): Solution approaches. `https://towardsdatascience.com/reinforcement-learning-made-simple-part-2-solution-approaches-7e37cbf2334e`. Acessado em 31/03/2022.

Fenjiro, Y. e Benbrahim, H. (2018). Deep reinforcement learning overview of the state of the art. *Journal of Automation, Mobile Robotics and Intelligent Systems*, páginas 20–39.

Glatt, R., Da Silva, F. L. e Costa, A. H. R. (2016). Towards knowledge transfer in deep reinforcement learning. Em *2016 5th Brazilian Conference on Intelligent Systems (BRACIS)*, páginas 91–96. IEEE.

Hill, A., Raffin, A., Ernestus, M., Gleave, A., Kanervisto, A., Traore, R., Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S. e Wu, Y. (2018). Stable baselines. `https://github.com/hill-a/stable-baselines`.

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

Lawrence, S., Giles, C. L., Tsoi, A. C. e Back, A. D. (1997). Face recognition: A convolutional neural-network approach. *IEEE transactions on neural networks*, 8(1):98–113.

Lazaridis, A., Fachantidis, A. e Vlahavas, I. (2020). Deep reinforcement learning: A state-of-the-art walkthrough. *Journal of Artificial Intelligence Research*, 69:1421–1471.

LeCun, Y., Bottou, L., Bengio, Y. e Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

Maquaire, N. (2020). Are the space invaders deterministic or stochastic? `https://towardsdatascience.com/are-the-space-invaders-deterministic-or-stochastic-595a30becae2`. Acessado em 08/09/2021.

Marvin, M. e Seymour, A. P. (1969). Perceptrons.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D. e Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. Em *International conference on machine learning*, páginas 1928–1937. PMLR.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. e Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G. et al. (2015). Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533.

Munos, R., Stepleton, T., Harutyunyan, A. e Bellemare, M. (2016). Safe and efficient off-policy reinforcement learning. *Advances in neural information processing systems*, 29.

Nagabandi, A., Kahn, G., Fearing, R. S. e Levine, S. (2018). Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning. Em *2018 IEEE International Conference on Robotics and Automation (ICRA)*, páginas 7559–7566. IEEE.

Nielsen, M. A. (2015). *Neural networks and deep learning*. Determination Press.

Olivas, E. S., Guerrero, J. D. M., Martinez-Sober, M., Magdalena-Benedito, J. R., Serrano, L. et al. (2009). *Handbook of research on machine learning applications and trends: Algorithms, methods, and techniques: Algorithms, methods, and techniques*. IGI global.

OpenAI (2018a). Key concepts in rl. `https://spinningup.openai.com/en/latest/spinningup/rl_intro.html`. Acessado em 08/09/2021.

OpenAI (2018b). Kinds of rl algorithms. `https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html`. Acessado em 08/09/2021.

Schulman, J., Levine, S., Abbeel, P., Jordan, M. e Moritz, P. (2015). Trust region policy optimization. Em *International conference on machine learning*, páginas 1889–1897. PMLR.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A. e Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Seita, D. (2016). Frame skipping and pre-processing for deep q-networks on atari 2600 games. `https://danieltakeshi.github.io/2016/11/25/frame-skipping-and-preprocessing-for-deep-q-networks-on-atari-2600-games/`. Acessado em 25/04/2022.

Shao, K., Tang, Z., Zhu, Y., Li, N. e Zhao, D. (2019). A survey of deep reinforcement learning in video games. *arXiv preprint arXiv:1912.10944*.

Simioni, T. (2018). An intro to advantage actor critic methods: let's play sonic the hedgehog! https://www.freecodecamp.org/news/an-intro-to-advantage-actor-critic-methods-lets-play-sonic-the-hedgehog-86d6240171d/. Acessado em 17/04/2022.

Skinner, B. F. (1938). The behavior of organisms: an experimental analysis.

Sutton, R. S. e Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

Torres, J. (2020). Deep q-network (dqn)-i. https://towardsdatascience.com/deep-q-network-dqn-i-bce08bdf2af. Acessado em 25/04/2021.

Wang, Z., Bapst, V., Heess, N., Mnih, V., Munos, R., Kavukcuoglu, K. e de Freitas, N. (2016a). Sample efficient actor-critic with experience replay. *arXiv preprint arXiv:1611.01224*.

Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M. e Freitas, N. (2016b). Dueling network architectures for deep reinforcement learning. Em *International conference on machine learning*, páginas 1995–2003. PMLR.

Weng, L. (2018). Policy gradient algorithms. https://lilianweng.github.io/posts/2018-04-08-policy-gradient/#acer. Acessado em 27/04/2022.

Wikipedia (2022). Markov decision process. https://en.wikipedia.org/wiki/Markov_decision_process. Acessado em 21/04/2022.

William John, T. (2010). Artificial intelligence - agents and environments.

Wu, Y., Mansimov, E., Liao, S., Radford, A. e Schulman, J. (2017). Openai baselines: Acktr and a2c. https://openai.com/blog/baselines-acktr-a2c/. Acessado em 17/04/2022.

APÊNDICE A – PARÂMETROS

A.1 ALGORITMOS

A Tabela A.1 apresenta os parâmetros utilizados para cada um dos algoritmos.

Tabela A.1: Parâmetros utilizados nos algoritmos.

	A2C	ACER	PPO2	DQN
policy	CnnPolicy	CnnPolicy	CnnPolicy	CnnPolicy
n_envs	16	16	8	1
gamma	0.99	0.99	0.99	0.99
n_timesteps	10M	10M	10M	10M
buffer_size	–	5000	–	10000
learning_rate	0.0007	0.0007	lin(0.00025, 0)	0.0004
vf_coef	0.25	0.5	0.5	–
ent_coef	0.01	0.01	0.01	–
cliprange	–	–	lin(0.1, 0)	–
batch_size	–	–	–	32
learning_starts	–	1000	–	10000
target_network _update_freq	–	–	–	1000
exploration_eps	–	–	–	lin(0.1, 0.01)

Em alguns parâmetros, é possível utilizar funções lineares para descrever parâmetros que evoluem a cada iteração, definido na tabela pela função *lin(valor_inicial, valor_final)*. Por exemplo, *lin(0.1, 0.01)* descreve um valor que decai de 0.1 para 0.001 em *n_timesteps* iterações.

Os valores são denotados por – quando não se aplicam para determinado algoritmo.

Os parâmetros utilizados nos algoritmos são descritos como:

- policy: Tipo de rede neural da política utilizada.
- n_envs: Quantidade de cópias do ambiente executando em paralelo.
- gamma: Fator de desconto sobre expectativas de recompensas futuras.
- n_timesteps: Número de iterações de treinamento.
- buffer_size: Tamanho do *buffer* de *experience replay*.
- learning_rate: Fator de aprendizagem.
- vf_coef: Coeficiente de peso para o valor (ou Q-valor no ACER) no cálculo do erro.
- ent_coef: Coeficiente de peso para o fator de entropia no cálculo do erro.
- cliprange: Limite de corte para a atualização da política do PPO2.
- batch_size: Tamanho do lote de amostras obtidas do *buffer* de *experience replay*.

- learning_starts: Quantidade de iterações executadas para coletar experiências antes do agente começar o aprendizado.
- target_network_update_freq: Frequência de atualização da rede alvo do DQN.
- exploration_eps: Probabilidade do agente tomar uma ação aleatória para explorar o ambiente. [1]

A.2 REDES NEURAIS CONVOLUCIONAIS

A arquitetura da CNN implementada é a mesma para todos os algoritmos, sendo a mesma proposta em (Mnih et al., 2015), contendo:

- Entrada contendo uma imagem de dimensão 84x84x4.
- Camada de convolução com 32 filtros de tamanho 8x8 com *stride* de tamanho 4, seguida de uma ativação RELU.
- Camada de convolução com 64 filtros de tamanho 4x4 com *stride* de tamanho 2, seguida de uma ativação RELU.
- Camada de convolução com 64 filtros de tamanho 3x3 com *stride* de tamanho 1, seguida de uma ativação RELU.
- Camada Totalmente Conectada com saída de tamanho 512.
- Camada de saída contendo uma saída para cada ação disponível no ambiente, podendo variar de 3 a 18.

[1] O parâmetro exploration_eps é uma combinação dos parâmetros exploration_fraction, exploration_initial_eps e exploration_final_eps definido apenas neste texto para facilitar a leitura.