UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE MATEMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

TIAGO CARVALHO GOMES MONTALVÃO

APLICANDO MODELOS DE APRENDIZADO POR REFORÇO PROFUNDO EM UM JOGO ADVERSÁRIO

RIO DE JANEIRO
2021

TIAGO CARVALHO GOMES MONTALVÃO

APLICANDO MODELOS DE APRENDIZADO POR REFORÇO PROFUNDO EM UM JOGO ADVERSÁRIO

Trabalho de conclusão de curso de graduação apresentado ao Departamento de Ciência da Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Daniel Sadoc Menasché

RIO DE JANEIRO

2021

CIP - Catalogação na Publicação

M763a

Montalvão, Tiago Carvalho Gomes
Aplicando Modelos de Aprendizado por Reforço Profundo em um Jogo Adversário / Tiago Carvalho Gomes Montalvão. -- Rio de Janeiro, 2021.
49 f.

Orientador: Daniel Sadoc Menasché.
Trabalho de conclusão de curso (graduação) - Universidade Federal do Rio de Janeiro, Instituto de Matemática, Bacharel em Ciência da Computação, 2021.

1. Aprendizado por Reforço Profundo. 2. Aprendizado de Máquina. 3. Jogos com Aposta. I. Menasché, Daniel Sadoc, orient. II. Título.

Elaborado pelo Sistema de Geração Automática da UFRJ com os dados fornecidos pelo(a) autor(a), sob a responsabilidade de Miguel Romeu Amorim Neto - CRB-7/6283.

TIAGO CARVALHO GOMES MONTALVÃO

APLICANDO MODELOS DE APRENDIZADO POR REFORÇO PROFUNDO EM UM JOGO ADVERSÁRIO

Trabalho de conclusão de curso de graduação apresentado ao Departamento de Ciência da Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 17 de junho de 2021

BANCA EXAMINADORA:

Daniel Sadoc Menasche

Prof. Daniel Sadoc Menasché

Participação por videoconferência.

Prof. João Carlos Pereira da Silva

Participação por videoconferência

Prof. João Antonio Recio Paixão

Participação por videoconferência.

Prof. Wouter Caarls

RESUMO

Este trabalho tem como objetivo explorar a área de aprendizado por reforço profundo, que tem apresentado grandes evoluções nos últimos anos e é considerada por diversos pesquisadores a área mais próxima da chamada Inteligência Artificial Geral ou Inteligência Artificial Forte. Uma variação do jogo da velha é apresentada e um modelo é proposto para o treinamento de um agente inteligente capaz de jogar este jogo, de tal forma que as ações do agente combinam uma parte numérica com uma parte categórica. Para isto, este trabalho introduz grande parte da modelagem encontrada na área de Aprendizado por Reforço Profundo. São exploradas diversas configurações para o treinamento deste agente a fim de validar a que apresenta o melhor desempenho no jogo. A pontuação é calculada com base em partidas contra um agente proposto ao longo do trabalho que age em partes de forma aleatória e em partes de forma inteligente. Por fim, são apresentados os resultados obtidos nos experimentos, desafios encontrados e propostas de melhoria para trabalhos futuros.

Palavras-chave: aprendizado por reforço. aprendizado de máquina. aprendizado profundo. rede neural.

ABSTRACT

This work aims to explore the area of Deep Reinforcement Learning, which has shown major developments in recent years and is considered by many researchers the closest area to the so-called Artificial General Intelligence. A variation of the tic-tac-toe game is introduced and a model is proposed for training an intelligent agent capable of playing this game, such that the agent's actions combine a numerical part with a categorical one. For this, this work introduces much of the modeling found in the area of Deep Reinforcement Learning. Several configurations for the training of the agent are explored to validate the one that presents the best performance in the game. The agent score is calculated based on matches against an agent, introduced throughout this work, that acts in parts at random and in parts intelligently. Finally, the results obtained in the experiments, challenges faced and proposals for improvement for future work are presented.

Keywords: reinforcement learning. machine learning. deep learning. neural network

LISTA DE ILUSTRAÇÕES

Figura 1 – Recriação do sistema MENACE 10
Figura 2 – Exemplos de ambientes do OpenAI Gym 11
Figura 3 – Exemplos de ambientes MuJoCo 12
Figura 4 – Visualização do jogo em um terminal 13
Figura 5 – Interação de um agente com um ambiente 15
Figura 6 – Ilustração de uma rede neural 21
Figura 7 – Comparação entre *Q-Learning* e *Deep Q-Learning* 24
Figura 8 – Ilustração da estrutura de *Actor-Critic* do DDPG 27
Figura 9 – Comparativo entre DQN, DDPG e modelo proposto 29
Figura 10 – Diagrama da rede proposta do ator 29
Figura 11 – Diagrama da rede proposta do crítico 30
Figura 12 – Pontuação do melhor agente ao longo dos episódios 34
Figura 13 – Função de custo do melhor agente ao longo dos episódios 35
Figura 14 – Valores do crítico para a primeira jogada 37

LISTA DE QUADROS

Quadro 1 – Comparativo de hiperparâmetros e performance por versão 33

LISTA DE ABREVIATURAS E SIGLAS

DQN	Deep Q-Network
DDPG	Deep Deterministic Policy Gradient
GPU	Graphics Processing Unit
MDP	Markov Decision Process
RAM	Random Access Memory
TD	Temporal Difference
VRAM	Video RAM

LISTA DE SÍMBOLOS

$\|A\|$	Cardinalidade do conjunto A
$\mathbb{R}$	Conjunto dos números reais
s_t	Estado do agente no instante de tempo t
a_t	Ação realizada pelo agente no instante de tempo t
r_t	Recompensa imediata recebida pelo agente no instante de tempo t
G_t	Recompensa de longo prazo recebida pelo agente a partir do instante de tempo t
γ	Taxa de desconto de um MDP
α	Taxa de aprendizado de algoritmos baseados no gradiente descendente
θ	Conjunto dos parâmetros de um modelo
$\mathcal{L}$	Função de custo
$\nabla_\theta f$	Gradiente da função f em relação aos parâmetros θ
$\in$	Pertence
ε	Probabilidade de escolher uma ação aleatória em uma política ε-*greedy*
τ	Parâmetro do *soft update* utilizado pelo modelo DDPG
$\mathcal{N}$	Processo estocástico de Ornstein–Uhlenbeck

SUMÁRIO

1 INTRODUÇÃO 10
1.1 PROBLEMA EXPLORADO 12
1.2 OBJETIVOS DO TRABALHO 14

2 MODELAGEM 15
2.1 APRENDIZADO POR REFORÇO 15
2.1.1 Markov Decision Processes 16
2.1.2 Funções de Bellman 16
2.1.3 Q-Learning 18
2.2 APRENDIZADO PROFUNDO 20
2.2.1 Redes Neurais 20
2.2.2 Otimização da função de custo 21
2.3 APRENDIZADO POR REFORÇO PROFUNDO 23
2.3.1 Deep Q-Learning 23
2.3.2 Deep Deterministic Policy Gradient 26
2.4 ADAPTAÇÃO PARA O PROBLEMA 27

3 EXPERIMENTOS 31
3.1 AGENTE BASE PARA COMPARAÇÃO 31
3.2 CONFIGURAÇÃO DOS EXPERIMENTOS 31
3.3 RESULTADOS OBTIDOS 33
3.4 DESAFIOS 35

4 CONCLUSÃO 38
4.1 TRABALHOS FUTUROS 38

REFERÊNCIAS 40

APÊNDICE A – CONCEITOS BÁSICOS **47**
A.1 TIPOS DE APRENDIZADO DE MÁQUINA 47
A.1.1 Aprendizado supervisionado 47
A.1.2 Aprendizado não supervisionado 48
A.1.3 Aprendizado por reforço 49
A.2 CADEIAS DE MARKOV 49

1 INTRODUÇÃO

A área de aprendizado por reforço (SUTTON; BARTO, 2018) consiste em treinar um agente a partir de experiências e interações observadas em um ambiente, ao receber recompensas por cada ação feita. Esta área é fortemente inspirada na psicologia humana e, por isso, é considerada umas das áreas mais promissoras para alcançar a chamada Inteligência Artificial Forte (GOERTZEL, 01 Dec. 2014; ROCHA; COSTA; REIS, 2020), que consistiria em uma inteligência capaz de aprender qualquer tarefa que um ser humano também é capaz.

Ao longo do tempo, diversos avanços na área vêm sendo feitos. Uma das primeiras aplicações de aprendizado por reforço foi na resolução do jogo da velha, como por exemplo (MICHIE, 1961), em que foi desenvolvido um sistema mecânico com o auxílio de caixas de fósforo chamado de MENACE (*Matchbox Educable Noughts and Crosses Engine*), ilustrado na figura 1. Neste sistema, cada caixa de fósforo representa um certo estado do tabuleiro e possui uma certa quantidade de peças coloridas internamente, representando cada ação possível naquele estado. Em cada jogada, uma peça é retirada aleatoriamente e a ação correspondente é realizada. Caso o sistema ganhe ou empate o jogo, para cada caixa utilizada para realizar os movimentos, mais peças das cores utilizadas são adicionadas. Caso o sistema perca o jogo, as peças amostradas são removidas das caixas correspondentes, desincentivando estes movimentos de serem realizados novamente em jogos futuros. Este é um exemplo de aprendizado por reforço, em que o sistema interage com o jogo, dando recompensas positivas para jogadas com resultados ganhadores e de empate, e recompensas negativas para jogadas com resultados perdedores.

Figura 1 – Recriação do sistema MENACE

Diversos trabalhos foram desenvolvidos além do MENACE com foco no jogo da velha. Alguns exemplos: (GATTI; EMBRECHTS, 2013) explora o uso de redes neurais

para a resolução do jogo da velha e de uma variante sua; (FAUSSER; SCHWENKER, 2011) aplica um *ensemble* (comitê) de M agentes treinados e combinados para a decisão de qual ação tomar em cada passo para alguns jogos de tabuleiro, incluindo o jogo da velha; (STEEG; DRUGAN; WIERING, 2015) utiliza o método de diferenças temporais (discutido na seção 2.1.3) para explorar uma variação tridimensional do jogo da velha; e (WANG; EMMERICH; PLAAT, 2018) utiliza uma variação do Q-Learning (discutido também na seção 2.1.3) para treinar agentes que aprendem a jogar jogos de forma mais genérica, aprendendo a jogar vários jogos de uma vez, incluindo o jogo da velha.

Outros jogos também foram sendo explorados ao longo do tempo. Esta é uma área com grande interesse de pesquisa, pois é simples treinar agentes, com técnicas de aprendizado por reforço, a partir da interação com um simulador, com o objetivo de conseguir pontuações cada vez melhores. Exemplos são o treinamento de um agente para jogar diversos jogos do videogame Atari 2600 (MNIH et al., 2013), treinamento de um agente para jogar o jogo Gamão (TESAURO, 1995) ou mesmo um agente, conhecido como AlphaGo (SILVER et al., 2016), para jogar o famoso jogo Go.

Outras possíveis aplicações incluem o controle de veículos autônomos (KIRAN et al., 2021), em que o agente é responsável por controlar a direção do veículo a partir de sensores ao redor do mesmo; robótica (KOBER; BAGNELL; PETERS, 2013; GU et al., 2016), para o treinamento do controle de peças robóticas; e mercado financeiro (FISCHER, 2018; CHARPENTIER; ELIE; REMLINGER, 2020; YANG et al., 2020), criando estratégias de investimento ou de *trading* de ativos financeiros.

Para popularizar o estudo da área e o desenvolvimento de novos modelos, alguns ambientes com tarefas e interações prontas foram implementados, como os do OpenAI Gym (BROCKMAN et al., 2016), com exemplos na figura 2, e o MuJoCo (TODOROV; EREZ; TASSA, 2012), com exemplos na figura 3.

Figura 2 – Exemplos de ambientes do OpenAI Gym

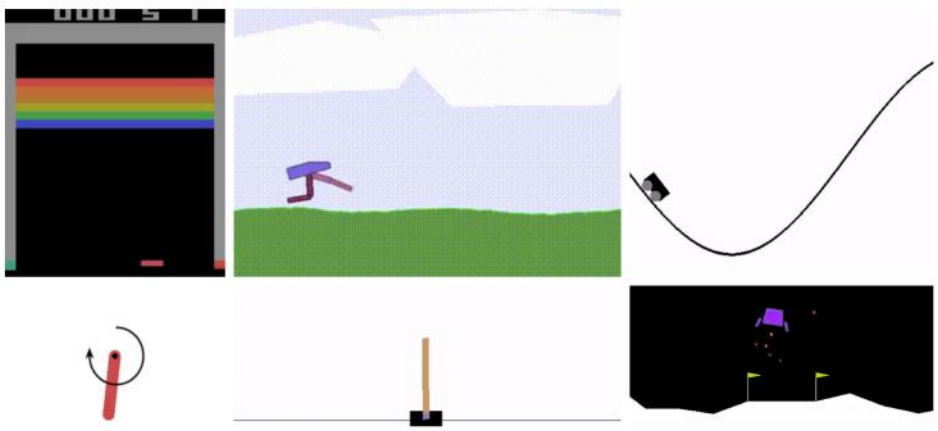
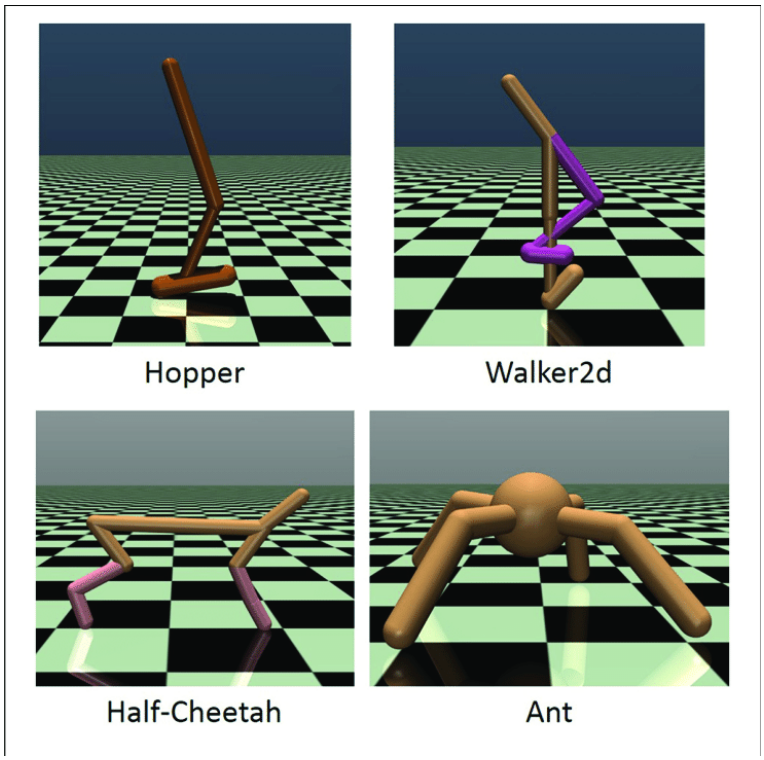


Figura 3 – Exemplos de ambientes MuJoCo



1.1 PROBLEMA EXPLORADO

O problema explorado ao longo deste trabalho é uma variação do famoso jogo da velha, conhecida como **Jogo da Velha com Aposta** (DEVELIN; PAYNE, 2010; FRANKO, 2019).

Nesta variação, dois jogadores competem em um tabuleiro idêntico ao do jogo da velha, que deve ser preenchido com três peças de um mesmo jogador na mesma linha, coluna ou diagonal. Porém, diferentemente da versão original, os jogadores não alternam turnos. Estas são as regras do jogo:

1. Cada jogador começa com uma quantidade C de moedas ($C = 100$ ao longo dos experimentos deste trabalho).

2. No começo de cada rodada, cada jogador deve apostar um valor entre 0 e o valor atual que possui de forma escondida, sem que seu oponente saiba este valor. Eles sabem apenas quantas moedas cada um possui.

3. As apostas são reveladas e o jogador que tiver o maior valor possui o direito de fazer um movimento no tabuleiro.

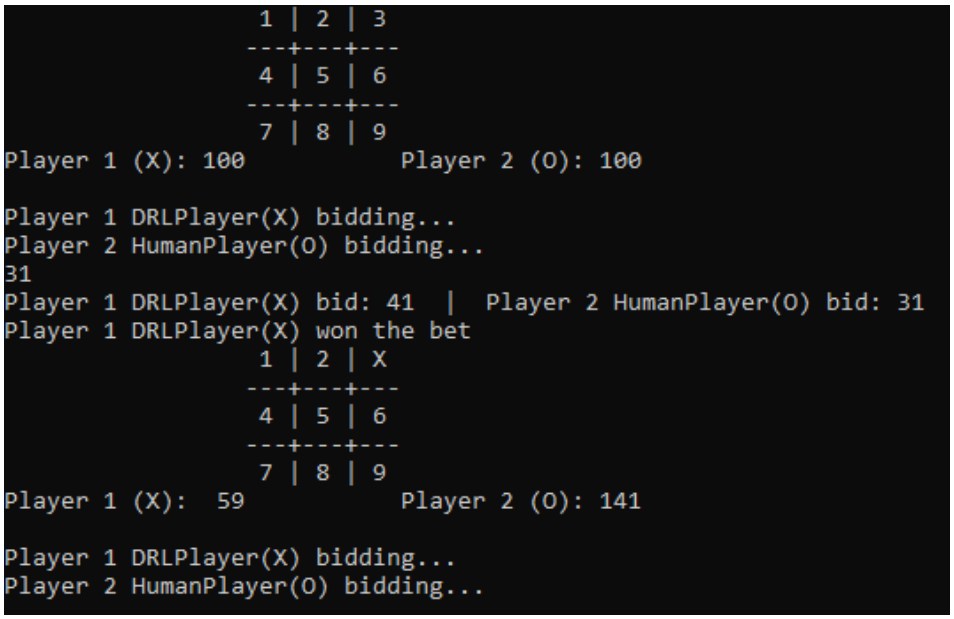
4. Este mesmo jogador deve dar o valor apostado para seu oponente.

5. As rodadas se seguem até um jogador possuir 3 peças em sequência, como no Jogo da Velha clássico.

Caso haja empate na aposta, o jogador que possuir mais moedas na rodada atual leva a vantagem. Caso ambos possuam a mesma quantidade, há um sorteio de qual jogador terá a vantagem na jogada.

Uma visualização do início deste jogo encontra-se na figura 4, no qual o jogador DRLPlayer apostou 41 moedas e ganhou a possibilidade de fazer um movimento no tabuleiro após ganhar a aposta contra seu oponente HumanPlayer, que apostou 31 moedas. Como cada jogador começou com 100 moedas, o jogador DRLPlayer agora possui 59 = 100 - 41 moedas, após fazer uma jogada na posição 3, enquanto o jogador HumanPlayer possui 141 = 100 + 41 moedas.

Figura 4 – Visualização do jogo em um terminal



Este jogo se mostra desafiador por alguns motivos. Estratégias do Jogo da Velha tradicional não se aplicam diretamente nesta variação, pois os jogadores não alternam turnos e, caso um jogador possua muitas moedas, há estratégias que o permitem garantidamente jogar pelo menos duas vezes seguidas, às vezes garantindo a vitória.

Além disso, há o elemento da surpresa: um jogador pode precisar apostar um valor alto para realizar certo movimento, e, caso seu oponente queira impedir tal movimento, precisa apostar um valor mais alto ainda. Mas o valor apostado pelo primeiro jogador pode ser baixo de propósito, fazendo com que ele receba muitas moedas de seu oponente e jogue com esta vantagem nas rodadas seguintes.

Alguns outros trabalhos nessa área de jogos com aposta foram realizados, como (DEVELIN; PAYNE, 2010), que segue uma abordagem combinatória, e (ARTHURS; BIRNBAUM, 2016), que também utiliza aprendizado por reforço profundo e segue uma metodologia similar à adotada neste trabalho, mas com uma modelagem diferente. O Jogo da Velha foi escolhido para este trabalho justamente por ser um dos jogos adversários mais simples

1.2 OBJETIVOS DO TRABALHO

Os principais objetivos deste trabalho são explorar parte da literatura do campo de aprendizado por reforço profundo, além de propor uma modelagem para o problema apresentado, junto com sua implementação e resultados obtidos.

No contexto deste jogo apresentado, se faz necessário um modelo capaz de processar um estado e retornar uma ação híbrida, com uma parte numérica (valor a ser apostado) e uma parte discreta (posição no tabuleiro). Alguns modelos existem com este propósito, como por exemplo o P-DQN (XIONG et al., 2018) e o modelo proposto em (WEI; WICKE; LUKE, 2018), em que o espaço de ações é parametrizado e a parte contínua é condicionada à parte discreta, ou seja, só é escolhida após a escolha da parte discreta e seu domínio pode variar em função desta.

Neste trabalho, é proposto mais um modelo, combinando características do modelo conhecido como DDPG (LILLICRAP et al., 2015) e de técnicas de exploração utilizadas no DQN (MNIH et al., 2013), para ser possível ter como saída uma ação com partes tanto discreta como contínua. Nesta proposta, os dois componentes (contínuo e discreto) são computados a partir de um mesmo modelo, como detalhado na seção 2.4. Isto é possível pois não há dependência entre o domínio do valor apostado com o domínio da posição escolhida no tabuleiro, isto é, o valor apostado não influencia em quais posições do tabuleiro podem ser jogadas e vice-versa. Esta modelagem faz com que haja um reaproveitamento das *features* calculadas a partir do estado atual (quantidade de moedas de cada jogador e configuração do tabuleiro) para a escolha de cada componente da ação a ser tomada.

A nível de experimentos, o objetivo consiste em treinar alguns agentes com este modelo proposto, variando alguns hiperparâmetros, e comparar seus resultados. Além disso, uma análise da qualidade das ações iniciais do melhor agente é apresentada, assim como métricas de treinamento e de validação.

Em aplicações reais, há a possibilidade de combinar modelos estatísticos com estratégias estabelecidas manualmente, que poderiam melhorar o desempenho do agente, mas estas não são exploradas neste trabalho.

Um exemplo clássico de estratégia desenvolvida por humanos é a *Tit For Tat* para o dilema do prisioneiro iterado (AXELROD, 1985). Em cada rodada, dois agentes interagem escolhendo uma ação cooperativa ou uma desertora. Cada combinação gera recompensas potencialmente diferentes para cada agente, de tal forma que, no longo prazo, é mais vantajoso para cada agente escolher a ação colaborativa. Foram realizados alguns torneios abertos para a comunidade, em que várias estratégias foram submetidas, e a conhecida como *Tit For Tat*, que consiste basicamente em começar colaborando e depois replicando a última ação do oponente, foi a que obteve melhor pontuação quando confrontada com todas as outras.

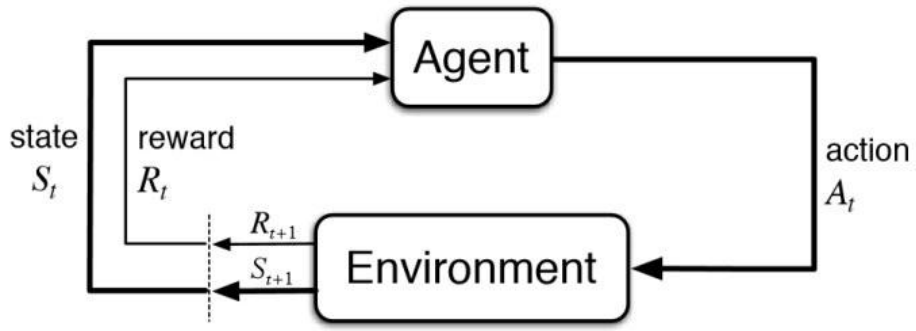
2 MODELAGEM

Para a criação de um agente capaz de realizar movimentos dentro do jogo, uma modelagem foi proposta utilizando conceitos de algumas áreas do aprendizado por reforço profundo. Inicialmente, foram utilizados modelos clássicos de aprendizado por reforço. Porém, estes modelos logo se tornam inviáveis com o aumento do tamanho dos espaços de estados e de ações do ambiente do jogo. Assim, é necessário modelar os estados e as ações de forma aproximada. Para isto, redes neurais são apresentadas, utilizadas e combinadas com técnicas anteriores. Por fim, modelos mais complexos são utilizados e adaptados para o Jogo da Velha com Aposta.

2.1 APRENDIZADO POR REFORÇO

Como introduzido anteriormente de forma breve, o **aprendizado por reforço** é uma área do aprendizado de máquina que consiste na interação de um agente com um ambiente. Este agente observa um estado no ambiente, toma uma ação, transiciona para um novo estado e recebe uma recompensa nesta transição. Uma visualização desta dinâmica encontra-se na figura 5.

Figura 5 – Interação de um agente com um ambiente



Agente observa estado S_t e recompensa R_t, toma uma ação A_t e vai para um novo estado S_{t+1} recebendo uma nova recompensa R_{t+1}.

O principal objetivo então de um agente treinado a partir de técnicas de aprendizado por reforço é o de encontrar uma política $\pi : \mathcal{S} \rightarrow \mathcal{A}$ que aprende qual ação $a \in \mathcal{A}$ tomar a partir de cada estado $s \in \mathcal{S}$ a fim de maximizar uma função objetivo baseada nas recompensas observadas.

Uma política pode ser **determinística**, na qual, para cada estado s, é associado exatamente uma ação $a = \pi(s)$, ou **estocástica**, na qual, para cada estado s, é associada uma distribuição de probabilidade $\pi(a|s)$ para todas as ações $a \in \mathcal{A}$.

Estas definições, assim como técnicas para encontrar tais políticas, são melhores discutidas nas seções a seguir.

2.1.1 Markov Decision Processes

Um *Markov Decision Process* (mais conhecido como MDP e traduzido como Processo de Decisão de Markov) (BELLMAN, 1957; HOWARD, 1960; SUTTON; BARTO, 2018) é uma formalização das noções de agente, estado, recompensa e ambiente descritos anteriormente, utilizando também a propriedade markoviana apresentada na seção A.2.

Nesta formulação, um agente interage com um ambiente em uma sequência de passos discretos. A cada passo t, o agente observa um estado $S_t \in \mathcal{S}$ do ambiente e toma uma ação $A_t \in \mathcal{A}(s)$. No passo seguinte $t+1$, ele recebe uma recompensa $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$ e observa um novo estado $S_{t+1} \in \mathcal{S}$.

Um MDP pode ser definido então como uma tupla $< \mathcal{S}, \mathcal{A}, p >$, em que:

- $\mathcal{S}$ é o conjunto de estados possíveis observados pelo agente
- $\mathcal{A}$ é o conjunto de ações possíveis a serem tomadas pelo agente
- $p(s', r|s, a)$ é a função de probabilidade, também chamada de **função de transição**, que define a dinâmica do MDP. Consiste na probabilidade de observar um novo estado $s' \in \mathcal{S}$ e uma recompensa $r \in \mathcal{R}$ no instante seguinte à tomada da ação $a \in \mathcal{A}$ no estado $s \in \mathcal{S}$, ou seja, é outro jeito de escrever $Pr(S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a)$

É comum, em algumas formulações de MDP, definir a função de transição sem a dependência das recompensas. A partir desta definição acima, poderíamos calcular esta nova função como uma distribuição marginal:

$$p(s'|s, a) = Pr(S_t = s'|S_{t-1} = s, A_{t-1} = a) = \sum_{r \in \mathcal{R}} p(s', r|s, a) \tag{2.1}$$

Conhecendo a função de transição do MDP, é possível calcular diversas outras informações, como o valor esperado da recompensa em uma transição entre estados.

2.1.2 Funções de Bellman

Dado um MDP, é necessário entender que função objetivo será maximizada. Para isso, é necessário entender a natureza do problema, se ele é **episódico**, ou seja, tem uma quantidade finita de passos por episódio (como por exemplo o jogo proposto neste trabalho), ou **infinito**, ou seja, potencialmente a interação do agente com o ambiente pode não ter fim (como por exemplo um robô aprendendo a andar).

Para um problema episódico, pode-se definir o retorno esperado a partir do instante t e para um instante terminal T como:

$$G_t = R_{t+1} + R_{t+2} + \cdots + R_T \tag{2.2}$$

Para um problema infinito, não existe instante terminal T e uma soma infinita facilmente diverge. Por isso, é introduzida uma constante denominada *taxa de desconto*, representada por γ, que pode ser usada para alterar a equação 2.2:

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \\ &= R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{(t+1)+k+1} = R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{2.3}$$

, no qual $0 \leq \gamma < 1$. Se as recompensas são finitas, isto é, $R_t < R_{\max}$, o valor de G_t também é finito, pois:

$$\begin{aligned} G_t &= \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \leq \sum_{k=0}^{\infty} \gamma^k R_{\max} \\ &= R_{\max} \sum_{k=0}^{\infty} \gamma^k = \frac{R_{\max}}{1-\gamma} \end{aligned} \tag{2.4}$$

É fácil ver que o valor de γ influencia o peso dado a cada recompensa no futuro. Valores de γ próximos de 0 fazem o agente olhar para instantes mais próximos e praticamente ignorar o futuro mais distante, enquanto valores de γ mais próximos de 1 fazem o agente considerar mais as recompensas futuras.

A partir da recompensa esperada G_t, é conveniente definir mais duas funções. Levando em consideração uma política π a ser seguida, a equação 2.5 é definida como **função de valor**, enquanto a equação 2.6 é definida como **função de ação-valor**.

$$v_\pi(s) = \mathbb{E}_\pi \left[G_t | S_t = s \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s \right], \forall s \in \mathcal{S} \tag{2.5}$$

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}_\pi \left[G_t | S_t = s, A_t = a \right] \\ &= \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a \right], \forall s \in \mathcal{S}, a \in \mathcal{A} \end{aligned} \tag{2.6}$$

Vale notar que a função de ação-valor pode ser calculada para qualquer par (s, a), mesmo que a não seja uma ação a ser tomada a partir da política escolhida π.

É possível escrever a função de valor de um estado em função dos estados seguintes, assim como a função de ação-valor para os estados e ações seguintes. Estas funções escritas

dessa forma recursiva são chamadas de **funções de Bellman**. A função de valor pode ser escrita como:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi \left[G_t | S_t = s \right] \\ &= \mathbb{E}_\pi \left[R_t + \gamma G_{t+1} | S_t = s \right] \\ &= \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s'} \sum_{r} p(s', r|s, a) \left[r + \gamma \mathbb{E}_\pi \left[G_{t+1} | S_{t+1} = s' \right] \right] \\ &= \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s', r} p(s', r|s, a) \left[r + \gamma v_\pi(s') \right], \forall s \in \mathcal{S} \end{aligned} \tag{2.7}$$

Um MDP possui uma função de valor e uma função de ação-valor ótimas, definidas como a própria função avaliada com a melhor política possível:

$$v_*(s) = \max_\pi v_\pi(s), \forall s \in \mathcal{S} \tag{2.8}$$

$$q_*(s, a) = \max_\pi q_\pi(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A} \tag{2.9}$$

2.1.3 Q-Learning

Um problema das equações de Bellman como apresentadas até o momento é que é necessário conhecer a função de transição p do MDP, o que na maior parte dos problemas não é verdade. Para resolver este problema, algumas abordagens que não requerem um modelo do ambiente são apresentadas. A primeira delas é utilizando métodos de Monte Carlo.

Seja G_t um valor observado para o retorno a partir do tempo t até o fim de um episódio e α uma constante indica o tamanho de um passo. É possível calcular uma estimativa $V(S_t)$ para $v_*(s)$ de forma iterativa a partir de uma média móvel exponencial para um número suficiente de episódios como:

$$V(S_t) \leftarrow \alpha G_t + (1 - \alpha) V(S_t) = V(S_t) + \alpha (G_t - V(S_t)) \tag{2.10}$$

Um problema imediato com este método é que é necessário esperar um episódio inteiro terminar para realizar atualizações para as estimativas das funções de valor. Quando o episódio é muito longo, isto pode ser muito ineficiente e, para problemas infinitos, isto é inviável.

Uma outra abordagem mais utilizada é a partir de um método chamado de *Temporal Difference* (SUTTON, 1988) (mais conhecido como TD e traduzido como Diferença Temporal). Este método possibilita a atualização da estimativa para a função de valor após cada instante t da interação entre agente e ambiente, e não após cada episódio, a partir do uso de uma técnica chamada *bootstrap*, em que uma estimativa é atualizada a partir de outras estimativas. A equação de atualização apresentada por este método é então:

$$V(S_t) \leftarrow V(S_t) + \alpha (R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) \tag{2.11}$$

A partir dessa abordagem, um dos principais algoritmos em Aprendizado por Reforço foi desenvolvido, chamado de ***Q-Learning*** (WATKINS, 1989; WATKINS; DAYAN, 1992). Ele utiliza uma atualização parecida com a apresentada na equação 2.11, mas com a atualização da estimativa $Q(S_t, A_t)$ para $q_*(s, a)$:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_t + 1, a) - Q(S_t, A_t) \right] \tag{2.12}$$

O algoritmo completo pode ser encontrado no Algoritmo 1. Ele utiliza uma forma de explorar o espaço de estados conhecida como ε-*greedy*, no qual, com probabilidade ε, uma ação aleatória é tomada e, com probabilidade $(1 - \varepsilon)$, a melhor ação calculada é tomada em cada instante.

Uma vez que $Q(s, a)$ é calculado para todas os estados e ações, a política pode ser calculada de forma determinística como:

$$\pi(s) = \max_a Q(s, a) \tag{2.13}$$

Algoritmo 1: Q-Learning

Entrada: Tamanho do passo $\alpha \in (0, 1]$ e $\varepsilon > 0$

Saída: $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}$

$Q(s, a)$ inicializado de forma arbitrária exceto para estados terminais, onde $Q(terminal, \cdot) = 0$

para *cada episódio* **faça**
- $S \leftarrow$ estado inicial
- **enquanto** S *não é terminal* **faça**
  - Escolha ação A a partir de S usando uma estratégia ε-*greedy* com Q
  - Realize ação A, observe recompensa R e próximo estado S'
  - $Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \gamma \max_a Q(S', a) - Q(S, A) \right]$
  - $S \leftarrow S'$
- **fim**

fim

Outros métodos foram propostos na literatura também utilizando esta abordagem com TD. Exemplos são SARSA (SUTTON; BARTO, 2018; RUMMERY; NIRANJAN, 1994) e *Expected* SARSA (SUTTON; BARTO, 2018), que tentam levar em consideração não só o melhor valor da função de ação-valor para escolha da próxima ação, mas também o método de exploração utilizado, mas tais métodos não são tão utilizados quanto o *Q-Learning*.

Vale comentar também que o *Q-Learning* é um método conhecido como ***off-policy***, no sentido em que ações podem ser tomadas a partir de qualquer política, mesmo uma que não utilize a função $Q(s, a)$ atual. Isto permite que este algoritmo seja utilizado para um agente aprender com experiências passadas e inclusive de outros agentes.

Um problema de todos estes métodos é que a função a ser calculada possui um domínio extremamente grande. Para a função Q, é necessário o cálculo de $|\mathcal{S}| \times |\mathcal{A}|$ valores. Por ser um método iterativo, cada valor deve ser atualizado um número suficiente de vezes para obter convergência. O número de iterações deste algoritmo facilmente explode, tornando-o inviável para tais cenários. Neste contexto, algumas técnicas devem ser utilizadas, seja para discretizar o espaço de estados e ações, como por exemplo as técnicas conhecidas como *Tile Coding* (SUTTON; BARTO, 2018) ou *Coarse Coding* (SUTTON, 1996; SUTTON; BARTO, 2018), ou para aproximar as funções calculadas a partir de estados e funções. Esta segunda opção é a mais utilizada, sendo comum utilizar modelos aproximativos baseados em redes neurais.

2.2 APRENDIZADO PROFUNDO

Esta é uma subárea do Aprendizado de Máquina, na qual redes neurais são utilizadas, mais especificamente redes neurais profundas, isto é, com pelo menos duas camadas.

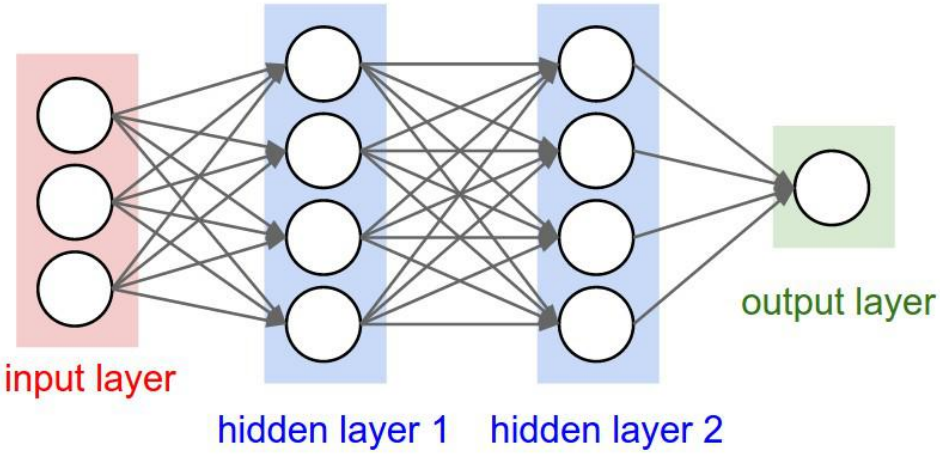
2.2.1 Redes Neurais

Uma rede neural (GOODFELLOW; BENGIO; COURVILLE, 2016) é um modelo paramétrico, criado a partir de fortes inspirações no cérebro humano. Este modelo consiste em um conjunto de unidades, chamadas de **neurônios**, conectadas entre si, formando um grafo de computação a partir de uma entrada. Estas conexões possuem parâmetros chamados de **pesos**.

É comum agrupar os neurônios em camadas, de tal forma que uma camada recebe sinais $\mathbf{x}$ ponderados pelos pesos $\mathbf{w}$ da camada anterior e combinados de forma linear em cada neurônio. Após esta combinação, é aplicada uma **função de ativação** não linear g a este sinal. É importante que esta função de ativação seja não linear, caso contrário a combinação de diversas funções lineares resultaria em uma única função linear. Uma visualização de uma rede neural simples em camadas pode ser vista na figura 6.

As redes neurais possuem diversos tipos de arquitetura, dentre elas redes *feed-forward*, convolucionais, recorrentes, entre outras. Cada rede costuma ser mais utilizada para um domínio de problemas. Por exemplo, redes convolucionais se tornaram muito conhecidas com a aplicação na classificação de imagens, assim como outras tarefas na área de visão computacional. Redes recorrentes possuem propriedades que a tornam adequadas para modelar entradas e/ou saídas sequenciais. Esta característica as torna apropriadas para a área de processamento de linguagem natural, expressa por textos escritos e comunicados entre seres humanos. O desenvolvimento de novas arquiteturas é uma área de pesquisa muito explorada, com o advento de redes cada vez mais complexas e potentes. Como exemplo, pode-se citar a criação das redes conhecidas como Transformers (VASWANI et

Figura 6 – Ilustração de uma rede neural



Exemplo de uma rede neural com três *features* de entrada, duas camadas ocultas e uma saída

al., 2017), muito utilizadas para modelagem de sequências, que possuem diversas vantagens sobre as redes recorrentes.

Embora as redes neurais fossem conhecidas há várias décadas, elas ficaram um tempo sem muita pesquisa ativa, enquanto não havia uma maneira eficiente de treiná-las, até o desenvolvimento do chamado ***backpropagation*** (RUMELHART; HINTON; WILLIAMS, 1986). Este é um algoritmo de propagação da função de custo em relação a cada um dos parâmetros, combinando técnicas de programação dinâmica com resultados de cálculo diferencial.

Mesmo com o desenvolvimento do *backpropagation*, treinar uma rede neural sempre foi considerado uma tarefa custosa. Por este motivo, este modelo tem se tornado cada vez mais utilizado nos últimos anos, com o poder computacional cada vez maior, tanto para processamento quanto para armazenamento, assim tornando conjuntos massivos de dados cada vez mais acessíveis.

Um exemplo de rede neural é a chamada AlexNet (KRIZHEVSKY; SUTSKEVER; HINTON, 2012), que foi a rede ganhadora da competição do ImageNet (DENG et al., 2009) na edição de 2012. Esta é uma competição em que um conjunto de 1,2 milhão de imagens deve ser classificado em 1000 classes distintas. A arquitetura da rede apresentava camadas convolucionais e ela foi treinada em GPU já naquela época.

2.2.2 Otimização da função de custo

Para o treinamento de uma rede neural, assim como vários outros modelos de aprendizado de máquina, é necessário otimizar uma função de custo que expressa o quão bem ela, parametrizada por um conjunto de parâmetros sendo ajustados, consegue representar a distribuição da saída em função da entrada.

Formalmente, dada uma função de custo $\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})$, é necessário minimizar seu valor, sabendo que $\hat{y} = \hat{f}(x; \theta)$, sendo θ é o conjunto de parâmetros da rede neural. Para otimizar esta função, diversas técnicas podem ser utilizadas, sendo a técnica do **gradiente descendente** a mais famosa.

Esta é uma técnica iterativa, na qual é calculado o gradiente da função de custo em relação a cada parâmetro e é feita uma atualização em cada um destes parâmetros. Este procedimento se repete um número máximo de vezes ou até a função de custo convergir para algum valor. Matematicamente, pode-se escrever esta atualização de forma vetorial como:

$$\theta_{t+1} = \theta_t - \alpha \nabla_{\theta_t} \mathcal{L}(\theta_t) \tag{2.14}$$

, no qual θ_t representa o conjunto de parâmetros na iteração t, α representa uma taxa de aprendizado, que indica o tamanho da atualização em cada iteração, e $\nabla_{\theta_t} \mathcal{L}(\theta_t)$ representa o gradiente de $\mathcal{L}$ quando avaliado com o conjunto de parâmetros θ_t em relação a cada um dos parâmetros de θ_t.

Na prática, atualização da equação 2.14 se torna um problema porque o cálculo de $\mathcal{L}$ requer o cálculos a partir de todos os pontos do conjunto de dados. Para contornar este problema, uma alternativa é utilizar o chamado **gradiente descendente estocástico**, que considera apenas um ponto do conjunto de dados por vez. Esta variação cria outro problema, que é a grande variabilidade das atualizações, devido às diferentes contribuições de cada ponto do conjunto de treinamento.

Para considerar as vantagens de ambas as abordagens, foi desenvolvida uma combinação destes métodos, conhecida como **gradiente descendente em mini-batch**, no qual é utilizada apenas uma amostra de um tamanho fixo como estimativa da função de custo para o conjunto todo. Esta alteração torna a atualização muito mais eficiente, porém em geral requer mais iterações para uma convergência melhor, além de introduzir ruídos na curva do custo ao longo das iterações.

Porém, mais um problema que surge é o de ajuste da taxa de aprendizado. Uma taxa de aprendizado muito pequena faz com que o ajuste dos parâmetros se dê de forma mais lenta, enquanto uma taxa de aprendizado muito grande faz com que o otimizador não consiga convergir para bons ótimos locais da curva da função de custo. Com esse problema em mente, algumas técnicas foram desenvolvidas para tornar a taxa de aprendizado adaptativa, entre elas: AdaGrad (DUCHI; HAZAN; SINGER, 2011), RMSProp (HINTON; SRIVASTAVA; SWERSKY, 2013) e Adam (KINGMA; BA, 2017).

O otimizador utilizado neste trabalho é o **Adam**. Ele utiliza dois parâmetros β_1 e β_2 para manter médias móveis exponenciais do gradiente e do quadrado do gradiente, respectivamente, e utiliza essas médias para automaticamente adaptar a taxa de aprendizado e para utilizar a ideia de **momento**, em que atualizações de iterações passadas influenciam na atualização atual do otimizador.

2.3 APRENDIZADO POR REFORÇO PROFUNDO

A partir das ideias discutidas, é possível combinar a modelagem de redes neurais com as ideias de aprendizado por reforço para calcular as funções de valor e de ação-valor de forma aproximada, dada uma entrada representando um estado ou um par de estado e ação.

2.3.1 Deep Q-Learning

A primeira ideia discutida é a de *Deep Q-Learning* (MNIH et al., 2013), que combina diretamente as ideias discutidas sobre *Q-Learning* com aproximações via redes neurais profundas, além de introduzir algumas técnicas para estabilizar o processo de treinamento. Esta ideia é utilizada pelos autores para a construção de uma rede neural convolucional para treinar agentes capazes de jogar diversos jogos de Atari 2600 sem que nenhuma regra do jogo fosse codificada, apenas com as interações do agente com o ambiente do jogo. Posteriormente, esta rede foi estendida e aplicada a 49 jogos de Atari 2600 (MNIH et al., 2015), atingindo performance comparável ou melhor que a de um humano em 29 destes jogos.

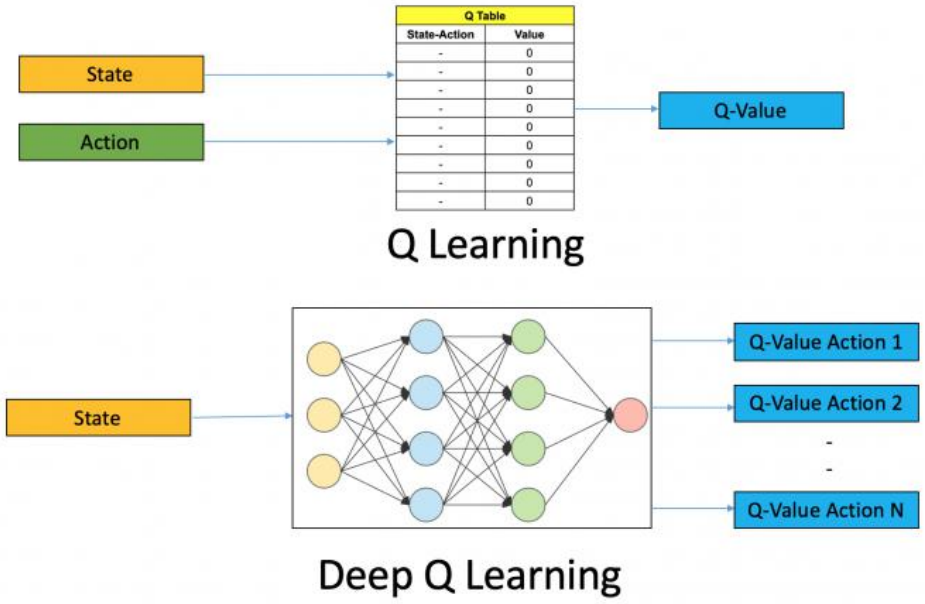
Recapitulando, para o conhecimento de $Q(s, a)$ para todo par de estado e ação, é necessário o cálculo de $|\mathcal{S}| \times |\mathcal{A}|$ posições. Neste problema específico, o tamanho do espaço de estados é dado por $|\mathcal{S}| = (2C + 1)3^{N^2} = 3956283$, sendo C a quantidade de moedas inicial de cada jogador (100 para os experimentos deste trabalho) e N o tamanho do lado do tabuleiro (3 para o tabuleiro clássico do jogo da velha), dado que um jogador pode ter entre 0 e $2C$ moedas e cada posição do tabuleiro pode estar vazia, preenchida pelo jogador 1 ou pelo jogador 2, enquanto o tamanho do espaço de ações se dá por até $|\mathcal{A}| = (2C+1)N^2 = 1809$, dado que um jogador pode apostar entre 0 e o número atual de moedas (que pode ser até $2C$) e fazer um movimento em qualquer posição do tabuleiro. Isto combinado resulta em mais de 7 bilhões de pares de estado e ação, tornando inviável a execução do algoritmo 1.

A ideia conhecida como *Deep Q-Learning* consiste em aprender uma função $Q(s, a; \theta)$, parametrizado por θ, como uma aproximação para $Q(s, a)$. A proposta apresentada pelo artigo que introduz esta ideia, e a utilizada neste trabalho, é realizar esta aproximação a partir de uma modelagem com uma rede neural profunda, conhecida então como *Deep Q-Network* (DQN). Um exemplo de um diagrama de uma DQN encontra-se na figura 7.

Após o cálculo dos valores aproximados para $Q(s, a; \theta)$, é utilizada alguma estratégia de escolha da ação a ser tomada pelo agente. Em geral, utiliza-se a mesma estratégia comentada na seção 2.1.3, conhecida como *ε-greedy*.

Uma ideia utilizada junto com o artigo foi a de ***Experience Replay*** (LIN, 1992), que consiste em um buffer de experiências passadas utilizado para o treinamento do modelo. Uma experiência consiste em uma tupla $< s_t, a_t, r_{t+1}, s_{t+1} >$, no qual s_t e a_t consistem

Figura 7 – Comparação entre *Q-Learning* e *Deep Q-Learning*



Comparação entre *Q-Learning*, em que todos os pares (s, a) são calculados, e *Deep Q-Learning*, em que, para cada estado s, são calculados os valores de $Q(s, a)$ para toda ação a possível. Fonte: https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/. Acesso em 04/03/2021.

em um estado e na ação tomada naquele estado em um determinado instante t, enquanto r_{t+1} e s_{t+1} consistem na recompensa recebida e no novo estado observado após a tomada da ação. Este buffer possui um certo tamanho B, de forma a armazenar as últimas B tuplas de experiência. Para o treinamento então, são amostradas algumas tuplas para o ajuste dos parâmetros. Esta ideia é útil para fazer o modelo aprender a partir de uma média de várias experiências passadas, evitando assim ficar preso nos conhecidos *feedback loops* e tornando o aprendizado mais suave.

Além do *Experience Replay*, foi introduzida também a ideia de ***Target Networks*** a partir de um problema observado de instabilidade no treinamento do modelo. A equação da função de custo pode ser expressa a partir de um erro quadrático médio (ver A.3), como apresentado pela equação 2.15, sendo θ o conjunto de parâmetros da rede:

$$\begin{aligned} \mathcal{L}(\theta) &= \mathbb{E}_{s,a,r,s'}\left[y - Q(s, a; \theta)\right]^2 \\ &= \mathbb{E}_{s,a,r,s'}\left[r + \gamma \max_{a'} Q(s', a'; \theta) - Q(s, a; \theta)\right]^2 \end{aligned} \tag{2.15}$$

Esta equação compara o valor atual calculado para $Q(s, a)$ com o valor calculado a partir de uma transição para um próximo estado, conhecido como *target* e calculado como $y = r' + \gamma \max_{a'} Q(s', a'; \theta)$. Esta abordagem possui um problema devido à natureza dinâmica do treinamento: quando uma atualização é feita em θ, não só o valor de $Q(s_t, a_t)$ muda, mas também o valor de $Q(s_{t+1}, a)$, para qualquer ação a, e, consequentemente, o

do target y. Sendo assim, foi introduzida uma segunda rede $\hat{Q}$ conhecida como *target network*, com arquitetura idêntica à rede original Q, com o intuito de se manter com seus parâmetros fixos, sendo atualizados a cada K iterações como uma cópia dos parâmetros da rede original.

A combinação de todos essas propostas encontram-se no algoritmo 2. Vale comentar que o algoritmo de otimização utilizado calcula o gradiente da função de custo em relação a θ, não a θ^-, portanto o target y_j é considerado constante para o treinamento.

A partir da versão inicial da DQN, surgiram diversas variações procurando atacar alguns problemas ainda encontrados. Por exemplo, os valores calculados para $Q(s, a; \theta)$ tendem a superestimar o valor real de $Q(s, a)$ (THRUN; SCHWARTZ, 1993), problema que motivou a variação conhecida como Double DQN (HASSELT; GUEZ; SILVER, 2015). Outro exemplo bem conhecido é a variação conhecida como Dueling DQN (WANG et al., 2016), em que é feita a decomposição $Q(s, a) = V(s) + A(s, a)$, sendo $A(s, a)$ um termo que indica a vantagem (*advantage*) de tomar uma ação a no estado s, quando comparado com as demais ações. Cada termo desses é aprendido de forma individual, resultando em melhores estimativas e melhores políticas em cenários em que ações diferentes possuem valores similares para a função de ação-valor.

Algoritmo 2: Deep Q-Learning

Buffer $\mathcal{D}$ de *experience replay* inicializado com tamanho B
Rede neural Q inicializada com parâmetros θ aleatórios
Rede neural *target* $\hat{Q}$ inicializada com parâmetros $\theta^- = \theta$
para *cada episódio* **faça**
  $s_1 \leftarrow$ estado inicial
  para $t = 1,\ T$ **faça**
    Escolha ação a_t a partir de s_t usando uma estratégia ε-*greedy* com Q
    Realize ação a_t, observe recompensa r_{t+1} e próximo estado s_{t+1}
    Guarde a tupla $< s_t, a_t, r_{t+1}, s_{t+1} >$ em $\mathcal{D}$
    Amostre tuplas de experiência $< s_j, a_j, r_{j+1}, s_{j+1} >$ de $\mathcal{D}$
    Faça $y_j = \begin{cases} r_{j+1} & \text{se episódio acaba em } j+1 \\ r_{j+1} + \gamma \max_{a'} \hat{Q}\left(s_{j+1}, a'; \theta^-\right) & \text{caso contrário} \end{cases}$
    Atualize θ com um passo do otimizador para $\left[y_j - Q\left(s_j, a_j; \theta\right)\right]^2$
    Faça $\hat{Q} = Q$ a cada K iterações
  fim
fim

2.3.2 Deep Deterministic Policy Gradient

Um problema de uma DQN é que ela, embora consiga aproximar o valor de $Q(s, a)$, as ações ainda precisam ser categóricas. Em um problema em que a ação é uma variável contínua, uma DQN não consegue ser bem utilizada. Para isso, alguma discretização no espaço de ações precisaria ser feito. Para contornar este problema, alguns modelos foram propostos, como por exemplo o Dyna-MLAC (COSTA; CAARLS; MENASCHÉ, 2015). Neste trabalho, é explorado o *Deep Deterministic Policy Gradient*, ou DDPG (LILLICRAP et al., 2015).

A ideia deste modelo é basicamente estender a DQN para permitir um espaço de ações contínuo, e, para fazer isso, o DDPG utiliza uma arquitetura ***Actor-Critic*** (BARTO; SUTTON; ANDERSON, 1983), em que dois modelos são treinados: um ator, responsável por aprender uma política $\mu(s; \theta^\mu)$ para o agente, isto é, quais ações devem ser tomadas em cada estado, e um crítico, responsável por avaliar a qualidade de um par de estado e ação, ao estimar $Q(s, a; \theta^Q)$ para a função de ação-valor. Estes dois modelos são ilustrados na figura 8.

O treinamento do DDPG é bem similar ao treinamento da DQN. Também é utilizada a técnica de *Experience Replay*, além das *Target Networks* para ambos ator e crítico.

O modelo do crítico é treinado utilizando o target y como uma estimativa do valor correto, assim como na DQN, e os parâmetros são ajustados a partir de uma função de custo de erro quadrático médio.

Já o ator é treinado com as estimativas do próprio crítico, a partir do gradiente da política (SUTTON et al., 2000). A rotina completa de treinamento encontra-se no algoritmo 3. Neste algoritmo, J representa a recompensa total esperada, já descontada com o fator de desconto γ. O cálculo de $\nabla_{\theta^\mu} J$ como o produto de dois gradientes na equação 2.16 é derivado no apêndice em (SILVER et al., 2014).

Uma forma mais simples de implementar a atualização dos parâmetros do ator é utilizar o negativo da saída do próprio crítico como função de custo. Desta forma, quanto maior o valor reportado pelo crítico, melhor é a ação retornada pelo ator, e menor deve ser a função de custo. De forma análoga, quanto menor o valor do crítico, pior é a ação retornada pelo ator, e maior deve ser a função de custo.

É introduzida uma nova estratégia de exploração, já que a estratégia *ε-greedy* é utilizada quando há um número finito de ações a serem escolhidas. Para um espaço de ações contínuo, o DDPG utiliza um processo estocástico $\mathcal{N}$ conhecido como Ornstein–Uhlenbeck (UHLENBECK; ORNSTEIN, 1930), que consiste em um passeio aleatório com uma certa tendência.

Mais uma diferença entre o DDPG e a DQN é a mudança da forma de atualizar as redes *target*. Na DQN, as redes *target* são atualizadas com os parâmetros da rede original a cada K iterações. As redes do DDPG são atualizadas em todas as iterações, mas a

Figura 8 – Ilustração da estrutura de *Actor-Critic* do DDPG

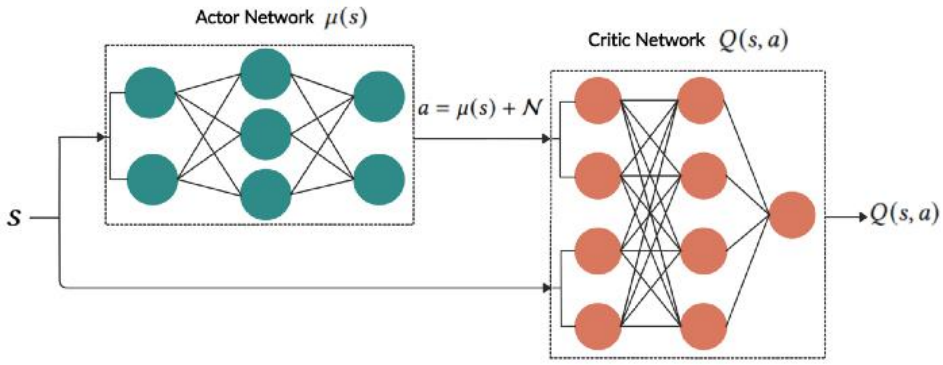


Ilustração da estrutura de *Actor-Critic* do DDPG. O ator gera uma ação a a partir do estado s, e é adicionado um ruído à ação realizada. Após tomar esta ação, o crítico avalia a qualidade desta ação a partir deste estado e essa informação é usada para o treinamento do ator. Fonte: http://finrl.org/algorithms/ddpg.html. Acesso em 04/03/2021.

partir de uma média móvel exponencial com parâmetro τ, com a equação de atualização expressa por $\hat{\theta} \leftarrow \tau\theta + (1-\tau)\hat{\theta}$, geralmente com $\tau \ll 1$.

2.4 ADAPTAÇÃO PARA O PROBLEMA

Para o problema do Jogo da Velha com Aposta, é necessário definir quais são os **estados** e as **ações** do jogo. Um estado é definido como $s = [c_1, c_2, b_1, b_2, \ldots, b_9]$, no qual c_i representa a quantidade de moedas do jogador i e $b_j \in \{0, 1, 2\}$ indica se a posição j do tabuleiro está vazia, quando $b_j = 0$, ou preenchida pela peça do jogador k, quando $b_j = k$, para $k \in \{1, 2\}$.

Uma ação poderia ser escrita como $a = [x, m]$, em que x indica o valor apostado e m indica a posição no tabuleiro a ser preenchida, caso a aposta seja vencedora.

Dada esta configuração, um modelo é então proposto, combinando o modelo DDPG com técnicas de exploração utilizadas no DQN (ε-*greedy*), para calcular a política e a função de ação-valor de forma aproximada, dado que parte da ação é numérica (valor a ser apostado) e parte é categórica (posição no tabuleiro). Um comparativo entre os 3 modelos pode ser encontrado na figura 9.

Seguindo então esta abordagem, um ator recebe o estado s e retorna uma ação a, como na figura 10, em que a aposta é um valor entre 0 e 1, representando uma porcentagem da quantidade atual de moedas do jogador a ser apostada, enquanto as demais saídas são probabilidades de fazer um movimento em cada posição do tabuleiro. Posições com peças já jogadas sempre recebem probabilidade 0. Estas saídas são então concatenadas em um vetor representando a ação computada.

Algoritmo 3: Deep Deterministic Policy Gradient

Buffer $\mathcal{D}$ de *experience replay* inicializado com tamanho B
Redes $Q(s, a; \theta^Q)$ e $\mu(s; \theta^\mu)$ inicializadas com parâmetros θ^Q e θ^μ aleatórios
Redes *target* $\hat{Q}(s, a; \theta^{\hat{Q}})$ e $\hat{\mu}(s; \theta^{\hat{\mu}})$ inicializadas com parâmetros $\theta^{\hat{Q}} = \theta^Q$ e $\theta^{\hat{\mu}} = \theta^\mu$
para *cada episódio* **faça**
  $s_1 \leftarrow$ estado inicial
  $\mathcal{N}$ processo aleatório para exploração
  para $t = 1,\ T$ **faça**
    Escolha ação $a_t = \mu\left(s_t; \theta^\mu\right) + \mathcal{N}_t$
    Realize ação a_t, observe recompensa r_{t+1} e próximo estado s_{t+1}
    Guarde a tupla $< s_t, a_t, r_{t+1}, s_{t+1} >$ em $\mathcal{D}$
    Amostre tuplas de experiência $< s_j, a_j, r_{j+1}, s_{j+1} >$ de $\mathcal{D}$
    Faça $y_j = \begin{cases} r_{j+1} & \text{se episódio acaba em } j+1 \\ r_{j+1} + \gamma \hat{Q}\left(s_{j+1}, \hat{\mu}(s_{j+1}; \theta^{\hat{\mu}}); \theta^{\hat{Q}}\right) & \text{caso contrário} \end{cases}$
    Atualize a rede do crítico minimizando $\mathcal{L} = \frac{1}{N} \sum_j \left[y_j - Q\left(s_j, a_j; \theta^Q\right)\right]^2$
    Atualize a rede do ator com a amostra obtida:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_j \nabla_a Q(s, a; \theta^Q) \mid_{s=s_j, a=\mu(s_j)} \nabla_{\theta^\mu} \mu\left(s; \theta^\mu\right) \mid_{s=s_j} \tag{2.16}$$

    Atualize as redes *target*:

$$\begin{aligned} \theta^{\hat{Q}} &\leftarrow \tau\theta^Q + (1-\tau)\theta^{\hat{Q}} \\ \theta^{\hat{\mu}} &\leftarrow \tau\theta^\mu + (1-\tau)\theta^{\hat{\mu}} \end{aligned} \tag{2.17}$$

  fim
fim

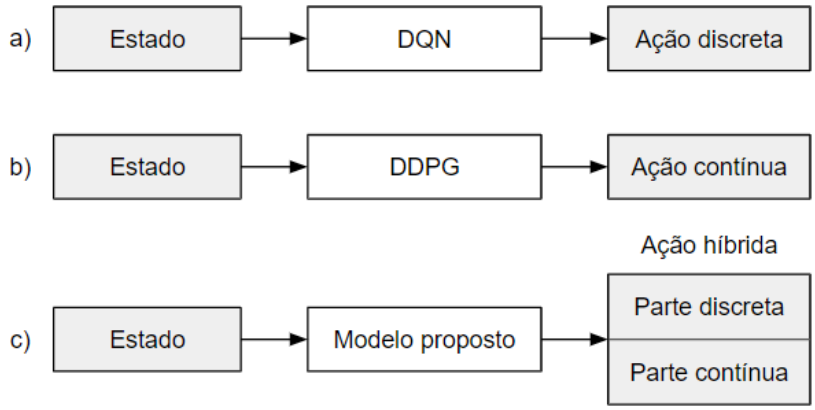
Esta forma de computar a ação a ser realizada, a partir de um único modelo para ambos componentes numérico (valor da aposta) e categórico (posição do tabuleiro), cria um reaproveitamento das *features* calculadas na última camada oculta da rede do ator, o que não seria o caso se cada componente fosse computado a partir de um modelo próprio.

Vale comentar que o ator, pelo modelo se basear no DDPG, aprende uma política determinística, isto é, sempre retorna a mesma ação a ser realizada em um determinado estado. Existem modelos como o *Soft Actor-Critic* (mais conhecido como SAC) (HAARNOJA et al., 2018) que possibilitam o aprendizado de uma política estocástica, em que é aprendida uma distribuição de probabilidade para as ações a partir de um estado. Algumas variações foram propostas ao longo dos últimos anos permitindo a modelagem de ações híbridas (CHRISTODOULOU, 2019; DELALLEAU et al., 2019), com parte contínua e parte discreta, mas esses modelos não foram explorados neste trabalho.

Já o crítico recebe um par de estado e ação e retorna um valor Q, representando a qualidade deste par, como ilustrado na figura 11.

Sobre a arquitetura das redes, ambas foram implementadas originalmente com 2 cama-

Figura 9 – Comparativo entre DQN, DDPG e modelo proposto



Comparativo entre os atores dos modelos: a) DQN, em que a saída do modelo é uma ação discreta, b) DDPG, em que a saída do modelo é uma ação contínua e c) o modelo proposto neste trabalho, em que a saída do modelo é uma ação híbrida, com um componente discreto e um componente contínuo.

Figura 10 – Diagrama da rede proposta do ator

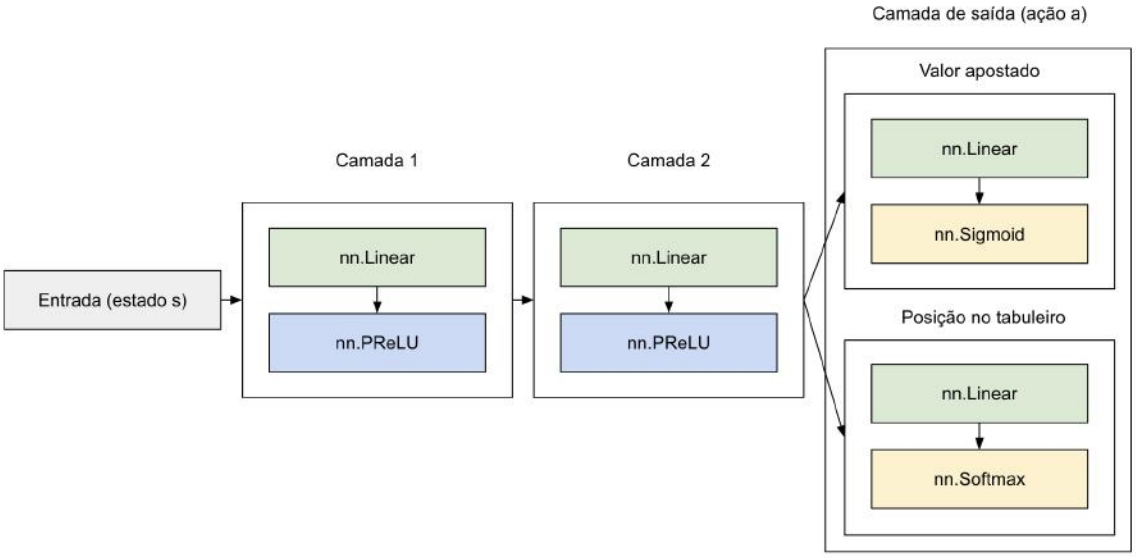


Diagrama da rede proposta para o ator do agente (tanto para a rede original quanto para a rede *target*). Nomes das camadas seguem o padrão do framework PyTorch.

das ocultas, cada uma contando com uma camada completamente conectada e uma função de ativação não-linear, neste caso uma *Parametric Rectified Linear Unit*, ou PReLU (HE et al., 2015), uma variação da função *Rectified Linear Unit*, ou ReLU (NAIR; HINTON, 2010), amplamente utilizada. Esta variação mostrou bons resultados com o pequeno custo de introduzir apenas mais 1 parâmetro por camada. O ator possui uma saída composta de duas partes, o valor apostado, implementado com uma função sigmoide, que retorna valores entre 0 e 1, e a posição no tabuleiro, implementada como uma *softmax*, em que

são atribuídas probabilidades a todos os possíveis resultados. No crítico, a saída é simplesmente uma camada completamente conectada, por sua saída ser um valor numérico.

A atualização das redes *target* é feita da mesma forma que no DDPG, em que é realizada de forma suave em cada iteração, a partir de uma média móvel exponencial com parâmetro τ.

A exploração de novas jogadas é uma combinação das estratégias de exploração de ambos DQN e DDPG, em que o processo estocástico $\mathcal{N}$, implementado com o processo Ornstein–Uhlenbeck, é utilizado para gerar perturbações na escolha da aposta, que é depois arredondada para o inteiro mais próximo, enquanto o ε-*greedy* é utilizado na escolha da posição do tabuleiro a ser jogada.

Em relação às **recompensas**, elas foram implementadas da seguinte forma: uma pequena recompensa negativa é dada a cada movimento, incentivando o agente a encerrar o jogo mais rápido, e uma recompensa com maior valor absoluto é dado no final do jogo caso o jogador ganhe (valor positivo) ou perca (valor negativo). Caso haja empate, a recompensa final é nula. Os valores exatos utilizados nos experimentos são apresentados na seção 3.2.

Figura 11 – Diagrama da rede proposta do crítico

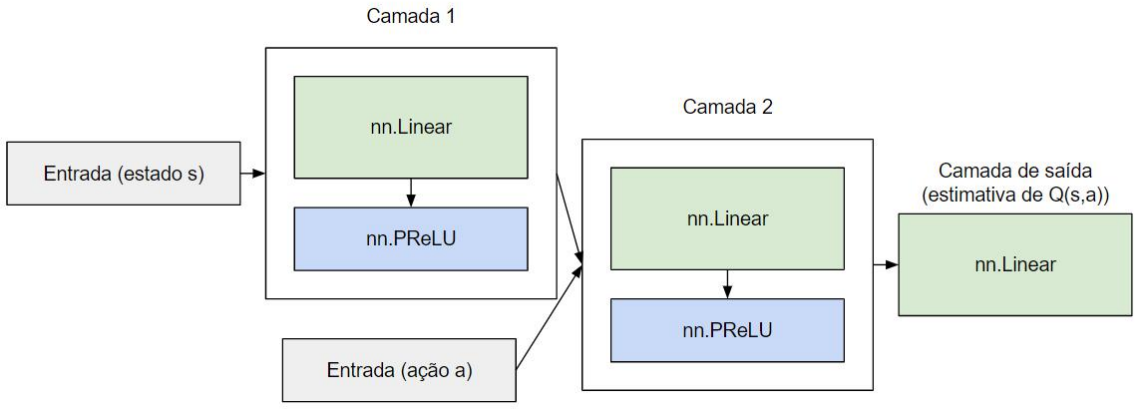


Diagrama da rede proposta para o crítico do agente (tanto para a rede original quanto para a rede *target*). Nomes das camadas seguem o padrão do framework PyTorch.

3 EXPERIMENTOS

3.1 AGENTE BASE PARA COMPARAÇÃO

Inicialmente um agente completamente aleatório foi implementado, tanto em relação ao valor apostado quanto em relação ao movimento realizado no tabuleiro caso a aposta fosse vencedora. A ideia deste agente era de ele ser diverso o suficiente para fornecer experiências variadas para o agente sendo treinado, de forma com que alguns cenários improváveis surgissem e gerassem tuplas de experiência para o agente reforçar o que fazer e o que não fazer.

Algumas versões do agente foram treinadas contra este agente aleatório, porém as métricas não eram muito boas, tanto a taxa de vitórias quanto o valor da função de custo dos modelos do ator e do crítico, que estabilizaram em uma época próxima do início e depois voltaram a aumentar, mesmo com valor menores da taxa de aprendizado.

Além disso, um comportamento foi observado: o agente inteligente treinado contra este agente aleatório começava jogando valores plausíveis, mas permitia que o oponente logo possuísse uma quantidade suficiente de moedas para garantidamente ganhar o jogo. Por exemplo, caso um jogador possua mais de 175 moedas, ele pode jogar 3 vezes seguidas com as seguintes apostas: 25, 50 e 100. Sendo assim, ele pode realizar movimentos preenchendo 3 casas em sequência no tabuleiro e ganhando o jogo. Da mesma forma, se o jogador possui mais de 150 moedas, ele pode garantidamente jogar pelo menos duas vezes seguidas, com as apostas 50 e 100, ganhando o jogo caso duas casas no tabuleiro estejam vagas em uma linha, coluna ou diagonal que já possua uma de suas peças.

Pensando nisso, foi desenvolvido um agente **semi-aleatório** que joga de forma aleatória no início, mas consegue identificar tais oportunidades de vitória garantida. Em tais casos, este agente aposta os valores exatos para conseguir jogar todas as rodadas até ganhar.

Este agente se mostrou muito mais rico no treinamento do agente por aprendizado por reforço, como será mostrado adiante, fazendo com que o agente treinado realizasse jogadas que levando em consideração não somente o instante atual, mas também as recompensas futuras do final do jogo.

3.2 CONFIGURAÇÃO DOS EXPERIMENTOS

Todo o código do projeto foi escrito na linguagem Python, utilizando o framework PyTorch (PASZKE et al., 2019) para a implementação e treinamento das redes neurais. O código está disponível publicamente no GitHub. [1]. Foi utilizada como base a implemen-

[1] https://github.com/tiagomontalvao/gambling-tic-tac-toe

tação do modelo DDPG fornecida no repositório de um curso online na área (UDACITY, 2019).

Foi utilizado para o treinamento a técnica de *gradient clipping* (PASCANU; MIKOLOV; BENGIO, 2013), na qual é atribuído um valor máximo para o módulo dos gradientes do erro em relação a cada parâmetro e, caso o módulo se torne maior, ele será atualizado para este valor máximo. Esta técnica evita um problema conhecido como gradiente explosivo, em que o gradiente da função de custo durante o treinamento se torna tão grande que começa a apresentar instabilidades numéricas e até fica impossível de armazenar em memória, motivada por redes neurais recorrentes, mas que apresenta bons resultados em outras arquiteturas (ZHANG et al., 2020).

Os modelos possuem um conjunto relativamente grande de hiperparâmetros a serem configurados. Alguns deles foram explorados, utilizando como base valores amplamente utilizados na literatura. O modelo final possui a seguinte configuração:

- **Otimizador utilizado:** Adam
- ***Learning rate* do otimizador:** 10^{-4}
- **Tamanho das redes neurais:** camadas com largura 256
- **Recompensa por movimento:** -0.05
- **Recompensa por ganhar/perder o jogo:** ± 1
- **Taxa γ de desconto do MDP:** 1.0
- **Tamanho do batch amostrado do *experience replay*:** 1024
- **Tamanho do buffer de *experience replay*:** 10^6
- **Valor do *gradient clipping*:** 5
- **Constante τ de atualização do modelo *target*:** 10^{-3}
- **Parâmetros do processo Ornstein-Uhlenbeck:** $\mu = 0$, $\theta = 0.2$ e $\sigma = 0.1$

Além disto, a estratégia ε-*greedy* foi utilizada, com um ε começando com o valor 1.0 e decaindo exponencialmente ao longo das épocas até o valor mínimo de 0.01, ou seja, no início do treinamento, 100% das posições escolhidas no tabuleiro são aleatórias, e esta porcentagem decai até chegar em 1%. Este parâmetro é zerado quando o agente encontra-se em modo de avaliação, ou seja, fora do treinamento nenhuma ação é aleatória.

Com esta arquitetura, ambos os modelos do ator e do agente possuem aproximadamente 128K parâmetros.

Todos os modelos dos agentes foram treinados em uma máquina com processador AMD Ryzen 7 3800X (3.9GHz de clock e 32MB de Cache) utilizando uma GPU RTX 2070 com 8GB de VRAM.

3.3 RESULTADOS OBTIDOS

Ao todo foram treinados 7 versões do agente, alterando algumas configurações de treinamento e hiperparâmetros. As configurações experimentadas foram:

- Agente oponente ao agente proposto treinado: aleatório / semi-aleatório / auto aprendizagem (contra si mesmo)
- Arquitetura da rede: quantidade de camadas e tamanho de cada camada
- Taxa de aprendizado: valores experimentados 10^{-3} e 10^{-4}
- Valor base para a ação do agente (o valor apostado quando o agente retorna 1 na saída da aposta do ator): quantidade atual de moedas do jogador ou total do jogo (200, neste caso)

Todos os modelos foram treinados por 100 mil épocas. Cada treinamento completo de 100 mil épocas levou entre 15 e 40 horas, com o maior tempo ocorrendo quando o agente foi treinado contra outro agente idêntico.

O quadro 1 apresenta um comparativo do desempenho dos 7 agentes treinados, com o resultado médio da execução de 10000 episódios para cada uma dessas versões, além da configuração de cada um desses agentes.

Quadro 1 – Comparativo de hiperparâmetros e performance por versão

MODELO	AGENTE OPONENTE	TAXA DE APRENDIZADO	ARQUITETURA (CAMADAS)	BASE DE MOEDAS	AGENTE ALEATÓRIO	AGENTE SEMI-ALEATÓRIO
v1	Idêntico	10^{-3}	(400, 300)	Qtd. atual	0.7043	0.5325
v2	Aleatório	10^{-3}	(400, 300)	Qtd. atual	0.8572	0.5465
v3	Idêntico	10^{-3}	(400, 300)	Total (200)	0.8475	0.5792
v4	Aleatório	10^{-3}	(400, 300)	Total (200)	0.9039	-0.0666
v5	Semi-aleatório	10^{-4}	(400, 300)	Total (200)	0.8453	0.7872
v6	**Semi-aleatório**	$\mathbf{10^{-4}}$	**(400, 300)**	**Qtd. atual**	**0.9203**	**0.8872**
v7	Semi-aleatório	10^{-4}	(256, 256, 256)	Qtd. atual	-0.6613	-0.7485

Configuração e performance de cada versão para os 7 agentes treinados. O modelo v6 foi o que apresentou melhor performance em ambos cenários, jogando tanto contra o agente aleatório como contra o agente semi-aleatório.

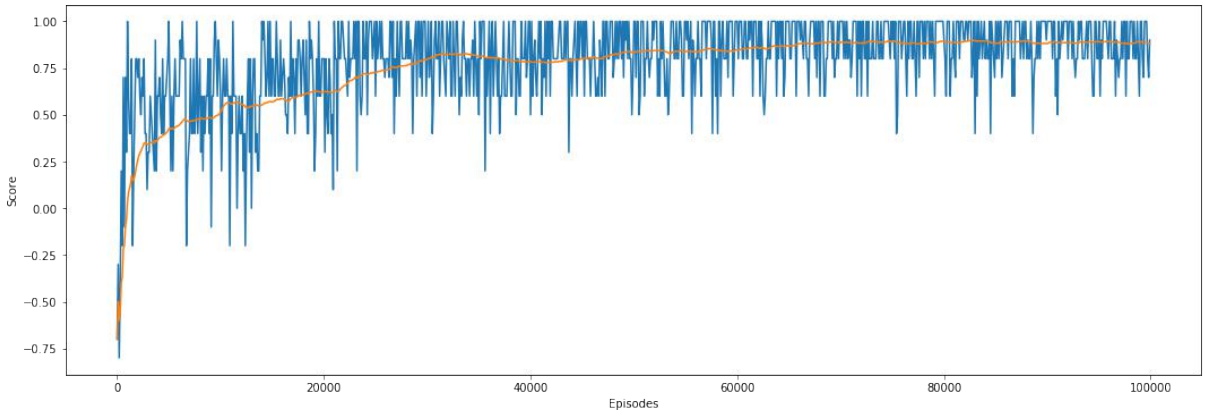
Segundo este quadro, o melhor modelo foi treinado contra o agente semi-aleatório, o que faz sentido, pois o agente aleatório não apresenta muita dificuldade como oponente no

jogo. A taxa de aprendizado foi variada entre modelos, mas não apresentou uma grande influência na performance final. A arquitetura da rede não foi tão explorada, e pode ser considerada como ponto de melhoria em trabalhos futuros. A base de moedas utilizada no melhor modelo é em relação à quantidade atual de moedas do jogador, isto é, um valor de 0.2 indica que a jogada é de 20% da quantidade atual, e não da quantidade total de moedas (200).

Vale comentar que, pela complexidade de tempo de treinar estes modelos, cada cenário foi treinado uma única vez, mas treinamentos posteriores poderiam dar resultados diferentes, pela característica estocástica do treinamento. Entretanto, pode-se observar que o melhor modelo é condizente com o esperado para alguns hiperparâmetros, como discutido acima.

A partir de agora, a versão (v6) que obteve melhor performance será analisada mais a fundo. Algumas métricas foram acompanhadas ao longo do treinamento. São elas: a pontuação do agente sendo treinado contra um agente semi-aleatório, como mostrado (v6) na figura 12, e as funções de custo de cada modelo (ator e crítico), como mostrado para o mesmo agente na figura 13.

Figura 12 – Pontuação do melhor agente ao longo dos episódios



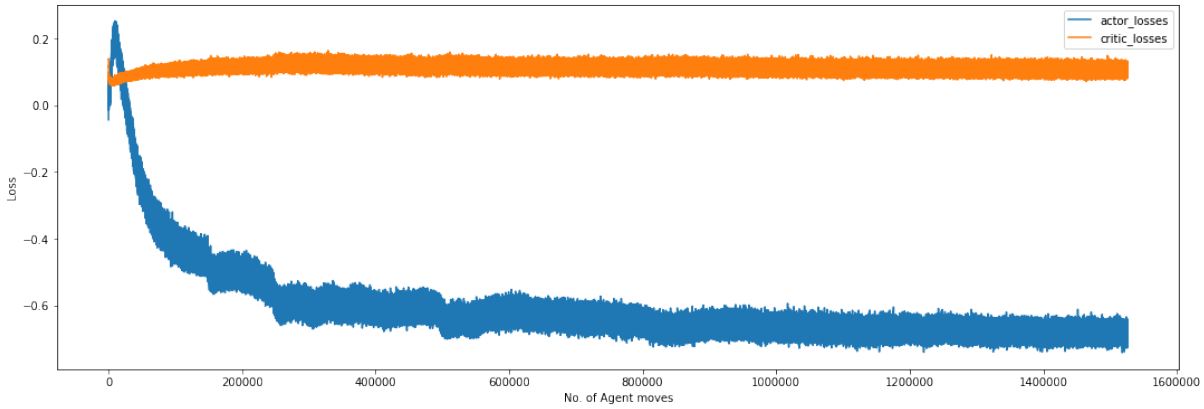
Pontuação do melhor agente ao longo dos episódios ao competir com um agente semi-aleatório (em azul) e média móvel de 10 pontos (em laranja)

O gráfico da figura 12 foi construído da seguinte forma: a cada 100 épocas de treinamento, o agente jogava contra o agente semi-aleatório 10 vezes. Cada vitória contribui +1 para a pontuação e cada derrota contribui com -1. Empates são neutros na pontuação. A pontuação dessa rodada de validação então é a média de todas as 10 partidas. Essas rodadas foram executadas ao longo do treinamento e representam o gráfico azul.

É possível observar a melhora na performance do agente ao longo dos episódios. No início, o agente perdia a maior parte dos jogos para o agente semi-aleatório, mas logo foi aprendendo com base nas suas experiências e foi capaz de ganhar de seu oponente na

maior parte das vezes.

Figura 13 – Função de custo do melhor agente ao longo dos episódios



Função de custo dos modelos do ator (em azul) e do crítico (em laranja) do melhor agente ao longo dos movimentos feitos durante o treinamento

O gráfico da figura 13 foi construído monitorando o valor das funções de custo de ambos ator e crítico ao longo do treinamento. A função de custo do crítico foi implementada a partir do MSE apresentado no algoritmo 3 e a função de custo do ator foi calculada como o negativo da saída do próprio crítico, assim como apresentado textualmente na seção 2.3.2.

Este gráfico apresenta um comportamento curioso nas primeiras atualizações dos modelos. O modelo do crítico se mantém com a função de custo praticamente constante ao longo do treinamento todo, embora esta função tenha caído bastante nas primeiras dezenas de atualizações, aumentou um pouco até o movimento 30000 e depois voltou a cair lentamente.

Já o modelo do ator apresentou um aumento brusco em seu valor para as primeiras centenas de movimentos do agente, mas depois seguiu caindo até o final do treinamento, momento em que continuava uma tendência de queda. É possível que este valor caísse mais caso o treinamento se prolongasse por mais tempo. Este aumento inicial não é desejado, mas reflete a dificuldade encontrada em treinar tais agentes, que, dependendo da configuração aplicada e dos hiperparâmetros escolhidos, pode apresentar uma instabilidade grande no treinamento.

3.4 DESAFIOS

Algumas versões do agente foram treinadas utilizando um conceito de auto aprendizagem, em que ele joga contra si mesmo. Porém, esta forma de treinamento acabou apresentando problemas na convergência do modelo e não se demonstrou uma boa alternativa para este problema. Talvez alguma configuração adicional de hiperparâmetros

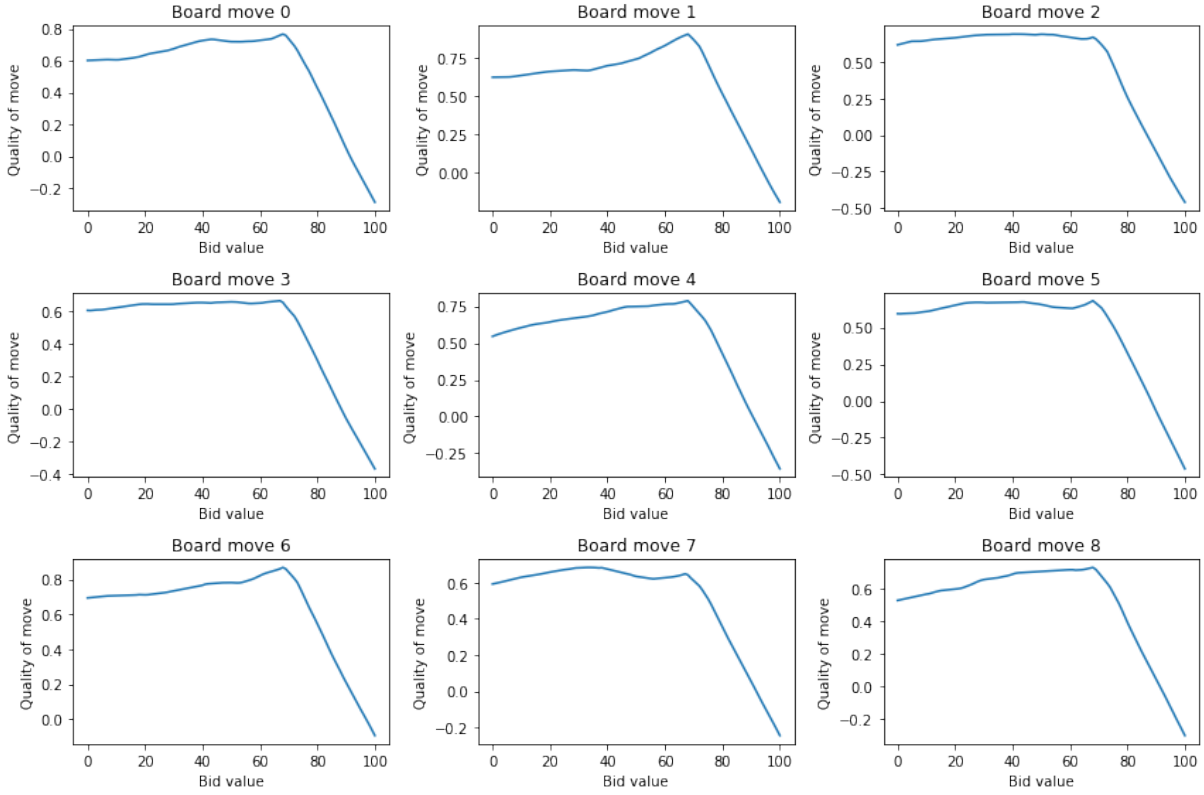
pudesse fazer este modo de treinamento ser mais bem sucedido, mas não foi o caso para os experimentos realizados.

A implementação deste agente é relativamente complexa, com vários detalhes a serem levados em consideração. Por conta disso, algumas performances de agentes treinados inicialmente estavam bem pobres devido a problemas de implementação encontrados, que foram consertados para os modelos apresentados no comparativo deste trabalho.

Uma questão a ser levantada é a velocidade baixa para iterar nas versões dos modelos. Como cada treinamento levava em torno de 1 dia completo, se tornou um problema fazer pequenas experimentações com hiperparâmetros. Por isso, eles não foram mais estressados do que o apresentado aqui. Caso o treinamento fosse mais rápido, seria mais fácil gerar mais versões dos modelos para seguir com a validação do melhor encontrado.

Um último desafio a ser levantado é a natureza do problema, que apresenta diferenças grandes na qualidade de uma jogada para variações pequenas na aposta feita. Essa falta de suavidade é um problema para a rede neural conseguir aproximar. Na figura 14 são apresentados os valores calculados no modelo do crítico para a primeira jogada no tabuleiro. Para cada possível primeira jogada no tabuleiro, é desenhado um gráfico com o valor de cada possível aposta. Neste caso, apostas muito grandes no início do jogo são desencorajadas. Isto ocorre a partir de um certo valor, por volta de 70, em que o valor calculado pelo crítico cai drasticamente.

Figura 14 – Valores do crítico para a primeira jogada



Valores do crítico para a primeira jogada possível no tabuleiro, considerando a posição no tabuleiro (cada gráfico) e a aposta (o eixo horizontal de cada gráfico)

4 CONCLUSÃO

Ao longo deste trabalho, o problema do **Jogo da Velha com Aposta** foi proposto e, para um agente poder ser treinado com técnicas de aprendizado por reforço profundo, foi necessário explorar uma gama grande de modelos e métodos.

Primeiro foi discutido um pouco do histórico da área de aprendizado de máquina, com foco em aprendizado por reforço. Depois, alguns conceitos básicos para entendimento da modelagem. Então, modelos e algoritmos de aprendizado por reforço e aprendizado profundo foram apresentados, junto com um modelo puramente estatístico proposto para o problema, isto é, sem regras fixas definidas manualmente, que deve levar em consideração uma ação híbrida, com parte categórica e parte numérica. Por último, os resultados dos agentes treinados foram discutidos junto com os desafios encontrados ao longo do trabalho.

4.1 TRABALHOS FUTUROS

Diversas melhorias podem ser implementadas ao longo das etapas do trabalho. A primeira delas é a experimentação dos modelos do agente com diferentes arquiteturas. Por exemplo, a forma como as moedas e o tabuleiro se relacionam na entrada das redes neurais do ator e do crítico poderia ser diferente. Uma possibilidade é adicionar uma outra camada para cada um desses componentes antes de combiná-los em uma camada posterior. Assim como poderiam haver mais camadas no final da rede neural do ator para cada componente (aposta e movimento no tabuleiro).

Outro ponto ainda sobre a arquitetura da rede a ser explorado é a inclusão de camadas convolucionais para processamento do tabuleiro. Por ser uma representação de um tabuleiro físico, existe uma noção de ordem espacial, que é bem capturada por operações de convolução.

Sobre a modelagem via aprendizado por reforço profundo, o modelo utilizado como base (DDPG) pode ser incrementado com técnicas sumarizadas no modelo Rainbow (HESSEL et al., 2017), como, por exemplo, *Experience Replay* com prioridades nas experiências, fazendo o agente atualizar seus parâmetros focando em tuplas de experiência mais relevantes, para alguma definição de relevância. Um outro modelo a ser experimentado é o *Twin Delayed DDPG* (mais conhecido como TD3) (FUJIMOTO; HOOF; MEGER, 2018), que é construído em cima do DDPG e utiliza algumas técnicas que melhoram sua estabilidade e performance.

Uma possibilidade de alteração no treinamento do agente é considerar a parte discreta da ação de fato como probabilidade, amostrando a posição jogada no tabuleiro a partir destas probabilidades. Esta estratégia pode substituir a exploração do agente via ε-*greedy*.

Enquanto o agente inteligente foi treinado jogando várias partidas contra um agente

chamado de semi-aleatório, uma outra possibilidade seria treinar contra um outro agente com os mesmos componentes (ator, crítico, etc), deixando com que a política de exploração proposta fizesse os agentes realizarem jogadas distintas. Um trabalho futuro seria o de realizar este auto-treinamento de forma adequada, fazendo com que resultados melhores potencialmente fossem alcançados, uma vez que o treinamento se daria contra um agente cada vez mais qualificado. Esta é a técnica utilizada pelos famosos AlphaGo Zero (SILVER et al., 2017), agente capaz de jogar o jogo Go, considerado mais difícil que o xadrez, e que fez com que ele conseguisse resultados melhores que o de humanos e o de agentes treinados anteriormente por outras técnicas, pelo AlphaZero, agente genérico capaz de jogar outros jogos além do Go, dado que nenhuma regra do jogo é codificada diretamente no agente, e pelo AlphaStar (VINYALS et al., 2019), agente capaz de jogar o jogo StarCraft II, considerado um dos jogos mais difíceis de estratégia em tempo real, com espaços de estados e de ações muito grandes (aproximadamente 10^{26} ações possíveis por instante). Estes agentes são sucessores do AlphaGo [1] (SILVER et al., 2016), que foi treinado utilizando como base movimentos de jogadores humanos experientes no Go, e joga contra outro agente idêntico a partir de certo instante.

Mesmo com todas estas alterações, o agente ainda possui um problema por aprender uma política completamente determinística, isto é, dado um estado do jogo, ele sempre irá realizar a mesma ação. Variações podem ser exploradas para introduzir um grau de aleatoriedade nas escolhas das ações, como o uso do modelo *Soft Actor-Critic*, que possui um ator que aprende uma política estocástica, e suas variações que permite a modelagem de ações híbridas.

Mais uma possibilidade de trabalho futuro seria o de estudar mais a fundo a dinâmica do problema para potencialmente chegar a soluções ótimas, assim como existem no jogo da velha tradicional em que um jogador ótimo nunca perde. A partir desta estratégia, seria possível inclusive treinar um agente jogando contra, de tal forma que as experiências seriam mais ricas e gerariam modelos melhores.

Por fim, seria interessante implementar um sistema online com o jogo e com o agente treinado, criando uma exposição do agente a outros jogadores com estratégias potencialmente diversas. Desta forma, o agente poderia continuamente ser treinado e melhorado. Seria possível também realizar um torneio com vários agentes, como a que revelou a estratégia *Tit For Tat* (AXELROD, 1985), para avaliar de forma mais assertiva a performance de cada um.

[1] A história do AlphaGo é documentada no filme disponível em https://www.alphagomovie.com/, acesso em 04/03/2021.

REFERÊNCIAS

ALTMAN, N. S. An introduction to kernel and nearest-neighbor nonparametric regression. **The American Statistician**, [American Statistical Association, Taylor & Francis, Ltd.], v. 46, n. 3, p. 175–185, 1992. ISSN 00031305. Disponível em: http://www.jstor.org/stable/2685209.

ARTHURS, N.; BIRNBAUM, S. Deep reinforcement learning for general game playing (theory and reinforcement). 2016.

AXELROD, R. **The Evolution of Cooperation**. [S.l.]: Basic Books, 1985. ISBN 0465021212,9780465021215,0465021220,9780465021222.

BARTO, A. G.; SUTTON, R. S.; ANDERSON, C. W. Neuronlike adaptive elements that can solve difficult learning control problems. **IEEE Transactions on Systems, Man, & Cybernetics**, Institute of Electrical & Electronics Engineers Inc, US, v. 13, n. 5, p. 834–846, 1983. ISSN 0018-9472(Print). Disponível em: https://doi.org/10.1109/TSMC.1983.6313077.

BELLMAN, R. A markovian decision process. **Indiana Univ. Math. J.**, v. 6, p. 679–684, 1957. ISSN 0022-2518.

BOTTOU, L.; BENGIO, Y. Convergence properties of the k-means algorithms. In: TESAURO, G.; TOURETZKY, D.; LEEN, T. (Ed.). **Advances in Neural Information Processing Systems**. MIT Press, 1995. v. 7. Disponível em: https://proceedings.neurips.cc/paper/1994/file/a1140a3d0df1c81e24ae954d935e8926-Paper.pdf.

BRIN, S.; PAGE, L. The anatomy of a large-scale hypertextual web search engine. In: **Proceedings of the Seventh International Conference on World Wide Web 7**. NLD: Elsevier Science Publishers B. V., 1998. (WWW7), p. 107–117.

BROCKMAN, G. et al. **OpenAI Gym**. 2016.

CHARPENTIER, A.; ELIE, R.; REMLINGER, C. **Reinforcement Learning in Economics and Finance**. 2020.

CHRISTODOULOU, P. **Soft Actor-Critic for Discrete Action Settings**. 2019.

CORTES, C.; VAPNIK, V. Support-vector networks. **Machine Learning**, v. 20, n. 3, p. 273–297, Sep 1995. ISSN 1573-0565. Disponível em: https://doi.org/10.1007/BF00994018.

COSTA, B.; CAARLS, W.; MENASCHÉ, D. Dyna-mlac: Trading computational and sample complexities in actor-critic reinforcement learning. In: . [S.l.: s.n.], 2015. p. 37–42.

DELALLEAU, O. et al. **Discrete and Continuous Action Representation for Practical RL in Video Games**. 2019.

DENG, J. et al. ImageNet: A Large-Scale Hierarchical Image Database. In: **CVPR09**. [S.l.: s.n.], 2009.

DEVELIN, M.; PAYNE, S. **Discrete bidding games**. 2010.

DUCHI, J.; HAZAN, E.; SINGER, Y. Adaptive subgradient methods for online learning and stochastic optimization. **J. Mach. Learn. Res.**, JMLR.org, v. 12, n. null, p. 2121–2159, jul. 2011. ISSN 1532-4435.

ESTER, M. et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In: **Proceedings of the Second International Conference on Knowledge Discovery and Data Mining**. [S.l.]: AAAI Press, 1996. (KDD'96), p. 226–231.

FAUSSER, S.; SCHWENKER, F. Ensemble methods for reinforcement learning with function approximation. In: SANSONE, C.; KITTLER, J.; ROLI, F. (Ed.). **Multiple Classifier Systems**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. p. 56–65. ISBN 978-3-642-21557-5.

FISCHER, T. G. **Reinforcement learning in financial markets - a survey**. Nürnberg, 2018. Disponível em: http://hdl.handle.net/10419/183139.

FIX, E.; HODGES, J. L. Discriminatory analysis. nonparametric discrimination: Consistency properties. **International Statistical Review / Revue Internationale de Statistique**, [Wiley, International Statistical Institute (ISI)], v. 57, n. 3, p. 238–247, 1989. ISSN 03067734, 17515823. Disponível em: http://www.jstor.org/stable/1403797.

FRANKO, O. **How would you modify the game Tic-tac-toe to make it more complex, but without changing the character of the game too much?** 2019. Quora. Acesso em 28/04/2021. Disponível em: https://qr.ae/pGhz5i.

F.R.S., K. P. Liii. on lines and planes of closest fit to systems of points in space. **The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science**, Taylor & Francis, v. 2, n. 11, p. 559–572, 1901. Disponível em: https://doi.org/10.1080/14786440109462720.

FUJIMOTO, S.; HOOF, H. van; MEGER, D. **Addressing Function Approximation Error in Actor-Critic Methods**. 2018.

GATTI, C. J.; EMBRECHTS, M. J. Reinforcement learning with neural networks: Tricks of the trade. In: _____. **Advances in Intelligent Signal Processing and Data Mining: Theory and Applications**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 275–310. ISBN 978-3-642-28696-4. Disponível em: https://doi.org/10.1007/978-3-642-28696-4_11.

GOERTZEL, B. Artificial general intelligence: Concept, state of the art, and future prospects. **Journal of Artificial General Intelligence**, Sciendo, Berlin, v. 5, n. 1, p. 1 – 48, 01 Dec. 2014. Disponível em: https://content.sciendo.com/view/journals/jagi/5/1/article-p1.xml.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. **Deep Learning**. [S.l.]: MIT Press, 2016. http://www.deeplearningbook.org.

GU, S. et al. **Deep Reinforcement Learning for Robotic Manipulation with Asynchronous Off-Policy Updates**. 2016.

HAARNOJA, T. et al. **Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor**. 2018.

HASSELT, H. van; GUEZ, A.; SILVER, D. **Deep Reinforcement Learning with Double Q-learning**. 2015.

HE, K. et al. **Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification**. 2015.

HESSEL, M. et al. **Rainbow: Combining Improvements in Deep Reinforcement Learning**. 2017.

HINTON, G.; SRIVASTAVA, N.; SWERSKY, K. **RMSProp: Divide the gradient by a running average of its recent magnitude**. 2013. Disponível em: http://www.cs.toronto.edu/~hinton/coursera/lecture6/lec6.pdf.

HOTELLING, H. Analysis of a complex of statistical variables into principal components. **Journal of Educational Psychology**, v. 24, p. 498–520, 1933.

HOWARD, R. A. Dynamic programming and markov processes. The M.I.T. Press, 1960.

KINGMA, D. P.; BA, J. **Adam: A Method for Stochastic Optimization**. 2017.

KIRAN, B. R. et al. **Deep Reinforcement Learning for Autonomous Driving: A Survey**. 2021.

KOBER, J.; BAGNELL, J.; PETERS, J. Reinforcement learning in robotics: A survey. **The International Journal of Robotics Research**, v. 32, p. 1238–1274, 09 2013.

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

LILLICRAP, T. et al. Continuous control with deep reinforcement learning. **CoRR**, 09 2015.

LIN, L.-J. Self-improving reactive agents based on reinforcement learning, planning and teaching. **Machine Learning**, Springer Science and Business Media LLC, v. 8, n. 3-4, p. 293–321, maio 1992. Disponível em: https://doi.org/10.1007/bf00992699.

LLOYD, S. Least squares quantization in pcm. **IEEE Transactions on Information Theory**, v. 28, n. 2, p. 129–137, 1982. Disponível em: https://app.dimensions.ai/details/publication/pub.1061648677andhttp://www.cs.toronto.edu/~roweis/csc2515-2006/readings/lloyd57.pdf.

MAATEN, L. van der; HINTON, G. Visualizing data using t-sne. **Journal of Machine Learning Research**, v. 9, n. 86, p. 2579–2605, 2008. Disponível em: http://jmlr.org/papers/v9/vandermaaten08a.html.

MICHIE, D. Trial and error. In: **On Machine Intelligence**. 1. ed. [S.l.]: Science Survey, 1961. v. 1.

MNIH, V. et al. **Playing Atari with Deep Reinforcement Learning**. 2013.

MNIH, V. et al. Human-level control through deep reinforcement learning. **Nature**, v. 518, n. 7540, p. 529–533, Feb 2015. ISSN 1476-4687. Disponível em: https://doi.org/10.1038/nature14236.

NAIR, V.; HINTON, G. E. Rectified linear units improve restricted boltzmann machines. In: **Proceedings of the 27th International Conference on International Conference on Machine Learning**. Madison, WI, USA: Omnipress, 2010. (ICML'10), p. 807–814. ISBN 9781605589077.

PASCANU, R.; MIKOLOV, T.; BENGIO, Y. On the difficulty of training recurrent neural networks. In: **Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28**. [S.l.]: JMLR.org, 2013. (ICML'13), p. III–1310–III–1318.

PASZKE, A. et al. Pytorch: An imperative style, high-performance deep learning library. In: WALLACH, H. et al. (Ed.). **Advances in Neural Information Processing Systems**. Curran Associates, Inc., 2019. v. 32. Disponível em: https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf.

ROCHA, F.; COSTA, V.; REIS, L. From reinforcement learning towards artificial general intelligence. In: _____. [S.l.: s.n.], 2020. p. 401–413. ISBN 978-3-030-45690-0.

RUMELHART, D. E.; HINTON, G. E.; WILLIAMS, R. J. Learning representations by back-propagating errors. **Nature**, v. 323, n. 6088, p. 533–536, Oct 1986. ISSN 1476-4687. Disponível em: https://doi.org/10.1038/323533a0.

RUMMERY, G.; NIRANJAN, M. On-line q-learning using connectionist systems. **Technical Report CUED/F-INFENG/TR 166**, 11 1994.

SILVER, D. et al. Mastering the game of go with deep neural networks and tree search. **Nature**, v. 529, n. 7587, p. 484–489, Jan 2016. ISSN 1476-4687. Disponível em: https://doi.org/10.1038/nature16961.

SILVER, D. et al. Deterministic policy gradient algorithms. In: **Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32**. [S.l.]: JMLR.org, 2014. (ICML'14), p. I–387–I–395.

SILVER, D. et al. Mastering the game of go without human knowledge. **Nature**, v. 550, n. 7676, p. 354–359, Oct 2017. ISSN 1476-4687. Disponível em: https://doi.org/10.1038/nature24270.

STEEG, M. van de; DRUGAN, M. M.; WIERING, M. Temporal difference learning for the game tic-tac-toe 3d: Applying structure to neural networks. In: **2015 IEEE Symposium Series on Computational Intelligence**. [S.l.: s.n.], 2015. p. 564–570.

SUTTON, R. Learning to predict by the method of temporal differences. **Machine Learning**, v. 3, p. 9–44, 08 1988.

SUTTON, R. S. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In: TOURETZKY, D.; MOZER, M. C.; HASSELMO, M. (Ed.). **Advances in Neural Information Processing Systems**. MIT

Press, 1996. v. 8. Disponível em: https://proceedings.neurips.cc/paper/1995/file/8f1d43620bc6bb580df6e80b0dc05c48-Paper.pdf.

SUTTON, R. S.; BARTO, A. G. **Reinforcement Learning: An Introduction**. Cambridge, MA, USA: A Bradford Book, 2018. ISBN 0262039249.

SUTTON, R. S. et al. Policy gradient methods for reinforcement learning with function approximation. In: SOLLA, S.; LEEN, T.; MüLLER, K. (Ed.). **Advances in Neural Information Processing Systems**. MIT Press, 2000. v. 12. Disponível em: https://proceedings.neurips.cc/paper/1999/file/464d828b85b0bed98e80ade0a5c43b0f-Paper.pdf.

TESAURO, G. Td-gammon: A self-teaching backgammon program. In: _____. **Applications of Neural Networks**. Boston, MA: Springer US, 1995. p. 267–285. ISBN 978-1-4757-2379-3. Disponível em: https://doi.org/10.1007/978-1-4757-2379-3_11.

THRUN, S.; SCHWARTZ, A. Issues in using function approximation for reinforcement learning. In: MOZER, M. et al. (Ed.). **Proceedings of the 1993 Connectionist Models Summer School**. [S.l.]: Erlbaum Associates, 1993.

TODOROV, E.; EREZ, T.; TASSA, Y. Mujoco: A physics engine for model-based control. In: **2012 IEEE/RSJ International Conference on Intelligent Robots and Systems**. [S.l.: s.n.], 2012. p. 5026–5033.

UDACITY. **Deep Reinforcement Learning Nanodegree**. 2019. Acesso em 04/03/2021. Disponível em: https://github.com/udacity/deep-reinforcement-learning.

UHLENBECK, G. E.; ORNSTEIN, L. S. On the theory of the brownian motion. **Phys. Rev.**, American Physical Society, v. 36, p. 823–841, Sep 1930. Disponível em: https://link.aps.org/doi/10.1103/PhysRev.36.823.

VASWANI, A. et al. **Attention Is All You Need**. 2017.

VINYALS, O. et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. **Nature**, v. 575, n. 7782, p. 350–354, Nov 2019. ISSN 1476-4687. Disponível em: https://doi.org/10.1038/s41586-019-1724-z.

WANG, H.; EMMERICH, M.; PLAAT, A. **Monte Carlo Q-learning for General Game Playing**. 2018.

WANG, Z. et al. **Dueling Network Architectures for Deep Reinforcement Learning**. 2016.

WATKINS, C. Learning from delayed rewards. 01 1989.

WATKINS, C. J. C. H.; DAYAN, P. Q-learning. **Machine Learning**, v. 8, n. 3, p. 279–292, May 1992. ISSN 1573-0565. Disponível em: https://doi.org/10.1007/BF00992698.

WEI, E.; WICKE, D.; LUKE, S. **Hierarchical Approaches for Reinforcement Learning in Parameterized Action Space**. 2018.

XIONG, J. et al. **Parametrized Deep Q-Networks Learning: Reinforcement Learning with Discrete-Continuous Hybrid Action Space**. 2018.

YANG, H. et al. **Reinforcement Learning for Automated Stock Trading: An Ensemble Strategy**. 2020. Disponível em https://ssrn.com/abstract=3690996 ou http://dx.doi.org/10.2139/ssrn.3690996.

ZHANG, J. et al. Why gradient clipping accelerates training: A theoretical justification for adaptivity. In: **International Conference on Learning Representations**. [s.n.], 2020. Disponível em: https://openreview.net/forum?id=BJgnXpVYwS.

APÊNDICES

APÊNDICE A – CONCEITOS BÁSICOS

A.1 TIPOS DE APRENDIZADO DE MÁQUINA

Toda a ideia do treinamento de um agente inteligente passa pela área de Inteligência Artificial. Neste trabalho, é explorada uma subárea conhecida como Aprendizado de Máquina, cujo principal objetivo é a combinação de modelos estatísticos com a aplicação de algoritmos para realizar o aprendizado de parâmetros capazes de aproximar distribuições de conjuntos de dados (modelos generativos) ou as saídas esperadas, dadas certas entradas (modelos discriminativos).

Dentro desta subárea, existem três grandes grupos de aprendizados, mais detalhados a seguir.

A.1.1 Aprendizado supervisionado

Este é o tipo mais comum de aprendizado. Nele, o objetivo é aprender uma função que mapeia entradas $x \in X$ em saídas $y \in Y$ a partir de um conjunto de exemplos de mapeamentos $\mathcal{D} =< (x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m) >$. Um exemplo de tarefa que pode ser resolvida com aprendizado supervisionado é o de identificar se uma imagem corresponde a um cachorro ou a um gato, ou então identificar qual o preço estimado de uma casa a partir de algumas características, chamadas ***features***, desta casa, como por exemplo o número de quartos, área construída, localização, entre outros.

A primeira tarefa de identificação de imagens em classes é um exemplo de uma tarefa de **classificação**, em que a saída esperada y é uma variável aleatória categórica.

Já a segunda tarefa de estimar o preço de uma casa é um exemplo de uma tarefa de **regressão**, em que a saída y é uma variável aleatória contínua, mesmo que possua algumas limitações de domínio (neste caso $y > 0$).

Tarefas de classificação e de regressão utilizam modelos muitas vezes similares, como por exemplo K vizinhos mais próximos (FIX; HODGES, 1989; ALTMAN, 1992), árvores de decisão e redes neurais (como visto na seção 2.2.1). Além destes modelos, existem outros muito utilizados, como SVM (CORTES; VAPNIK, 1995) e Regressão Logística para a tarefa de classificação, e Regressão Linear para a tarefa de regressão.

Seja qual for o tipo de tarefa, é necessário uma forma de aprender os melhores parâmetros do modelo que resolvem esta tarefa. Para isto, o problema de aprendizado supervisionado pode ser formulado como um problema de otimização, em que a função objetivo a ser minimizada é uma função de custo $\mathcal{L}(y, y^*)$, no qual $y = f(x)$ é o valor obtido a partir da função real a ser aprendida e $\hat{y} = \hat{f}(x)$ é o valor encontrado a partir da estimativa obtida para esta função.

A função mais utilizada para as tarefas de classificação é a **entropia cruzada**, calculada como:

$$l(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_i y_i \log \hat{y}_i \tag{A.1}$$

, no qual $\hat{\mathbf{y}}$ é o vetor de probabilidades calculadas pelo modelo e $\mathbf{y}$ é o vetor com todos os valores nulos, exceto pelo índice que indica a real classe daquele ponto. Para um conjunto de M exemplos, a função de custo é calculada como a média da entropia cruzada para cada exemplo:

$$\mathcal{L}_{\text{CE}}(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{M} \sum_{i=1}^{M} l(\mathbf{y}_i, \hat{\mathbf{y}}_i) \tag{A.2}$$

Para tarefas de regressão, a função mais utilizada para o cálculo da função de custo é o erro quadrático médio (também conhecido como MSE). Para um conjunto de M exemplos, esta função é calculada como:

$$\mathcal{L}_{\text{MSE}}(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{M} \sum_{i=1}^{M} (y_i - \hat{y}_i)^2 \tag{A.3}$$

A.1.2 Aprendizado não supervisionado

Enquanto no aprendizado supervisionado o conjunto de dados possui um rótulo para cada exemplo, isto é, um valor y associado, o aprendizado não supervisionado tem como objetivo encontrar padrões em conjuntos de dados da forma $\mathcal{D} =< x_1, x_2, \ldots, x_m >$, em que cada exemplo possui apenas um conjunto de *features*, porém sem rótulos.

As duas tarefas mais conhecidas são as tarefas de ***clustering*** e de **redução de dimensionalidade**.

A tarefa de *clustering* resume-se em encontrar grupos de exemplos similares no conjunto de dados. Por exemplo, para um conjunto de imagens de animais, uma tarefa possível seria a de agrupar estas imagens em grupos de animais semelhantes (sejam da mesma espécie, mesma família, mesma ordem, etc), sem que haja um conjunto de valores considerados corretos. Alguns exemplos de algoritmos de *clustering* são o K-Means (LLOYD, 1982; BOTTOU; BENGIO, 1995) e o DBSCAN (ESTER et al., 1996).

A tarefa de redução de dimensionalidade consiste em calcular um mapeamento $f : \mathbb{R}^n \rightarrow \mathbb{R}^k, k < n$, isto é, em uma projeção dos exemplos do conjunto de dados, originalmente com n dimensões, em um conjunto de dimensão k menor que n.

Alguns exemplos de algoritmos de redução de dimensionalidade são a Análise em Componentes Principais, ou PCA (F.R.S., 1901; HOTELLING, 1933), em que um mapeamento linear é computado de forma a maximizar a variância ao longo de cada eixo da base ortonormal calculada no espaço de dimensão menor, e o *t-Distributed Stochastic Neighbor Embedding*, ou t-SNE (MAATEN; HINTON, 2008), em que um mapeamento não-linear é calculado para esta projeção e é muito utilizado para gerar visualizações bidimensionais de espaços de dimensões maiores.

A.1.3 Aprendizado por reforço

Aprendizado por reforço é a terceira grande subárea, com uma grande diferença para as anteriores: diferente do aprendizado supervisionado, não existe um rótulo especificando qual a melhor ação a ser realizada em cada estado e, diferente do aprendizado não supervisionado, o objetivo não é modelar a distribuição de um conjunto de dados.

O aprendizado por reforço consiste em um agente interagindo com um ambiente e aprendendo a tomar ações a partir de observações e recompensas recebidas, com o objetivo de maximizar a soma destas recompensas ao longo do tempo. Esta subárea tem como inspiração a forma como os seres humanos (assim como outros animais) aprendem com experiências ao longo da vida e reforçam quais ações são boas e quais são ruins para serem tomadas em um determinado cenário.

Diversos modelos foram propostos, cada um para um tipo de problema, como por exemplo um espaço de ações contínuo ou discreto, computação de uma política de ação determinística ou estocástica, entre outros. Mais detalhes sobre as principais definições, os principais modelos e métodos são discutidos na seção 2.1.

A.2 CADEIAS DE MARKOV

A base da modelagem de aprendizado por reforço se dá a partir dos modelos markovianos. Um modelo com esta característica é tal que possui a propriedade markoviana, onde a probabilidade de um certo estado ocorrer depende somente do estado anterior e não do histórico de estados.

Mais formalmente, seja $\mathcal{S}$ um conjunto de estados, uma cadeia de Markov em tempo discreto é uma representação de um processo estocástico, no qual, dado um instante t, vale a propriedade:

$$P(X_t = s_t | X_{t-1} = s_{t-1}, \ldots, X_2 = s_2, X_1 = s_1) = P(X_t = s_t | X_{t-1} = s_{t-1}) \tag{A.4}$$

, no qual X_t é uma variável aleatória representando o estado observado no instante t e $s_1, s_2, \ldots, s_t \in \mathcal{S}$.

Existem algumas variações dessa definição, como a cadeia de Markov em tempo contínuo, no qual o tempo de transição entre estados é dado por uma variável aleatória de distribuição exponencial, que possui a propriedade de falta de memória, permitindo que o histórico de estados não interfira nas transições atual e futuras.

Cadeias de Markov possuem inúmeros usos, um dos mais famosos sendo no algoritmo Pagerank (BRIN; PAGE, 1998) que foi a base do mecanismo de busca do Google.