Pontifícia Universidade Católica
do Rio de Janeiro

Thomas Addis Junqueira Botelho

Extração de Políticas de Agentes Especialista para Árvores de Decisão Interpretáveis via Aprendizado por Imitação

Monografia de Projeto Final - Graduação

Trabalho apresentado como requisito parcial para obtenção de Bacharelado pelo Programa de Graduação em Informática, do Departamento de Informática da PUC-Rio .

Orientador: Prof. Augusto Cesar Espíndola Baffa

Rio de Janeiro
Julho de 2024

Thomas Addis Junqueira Botelho

Ficha Catalográfica

Botelho, Thomas

Extração de Políticas de Agentes Especialista para Árvores de Decisão Interpretáveis via Aprendizado por Imitação / Thomas Addis Junqueira Botelho; orientador: Augusto Cesar Espíndola Baffa. – 2024.

50 f: il. color. ; 30 cm

Projeto Final (Graduação) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2024.

Inclui bibliografia

1. Informática – Teses. 2. Aprendizado de Máquina. 3. Aprendizado Profundo. 4. Aprendizado por Imitação. 5. Redes Neurais. 6. DQN. 7. DDQN. 8. VIPER. 9. Árvores de Decisão. 10. Modelos Lineares. 11. Políticas Interpretáveis. 12. Automação de Tarefas. 13. Agentes Autônomos. 14. Simulação Ambiental. 15. Estabilidade Operacional. 16. Generalização de Modelos. I. Baffa, Augusto. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Resumo

Botelho, Thomas; Baffa, Augusto. **Extração de Políticas de Agentes Especialista para Árvores de Decisão Interpretáveis via Aprendizado por Imitação**. Rio de Janeiro, 2024. 50p. Relatório de Projeto Final II – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Este estudo apresenta uma investigação sobre a aplicação de técnicas de Aprendizado por Imitação (IL) para a extração de políticas estruturadas e interpretáveis a partir de modelos especialistas operando como caixas-pretas. O foco principal é analisar a viabilidade e eficácia dessa abordagem em traduzir comportamentos aprendidos por redes neurais profundas em árvores de decisão, que representam um conjunto de regras que podem ser sequencialmente avaliadas para chegar numa decisão. Avaliamos esta metodologia em três ambientes de simulação distintos: Lunar-Lander, Taxi e CartPole. Testamos os algoritmos DAgger e sua variante VIPER, que iterativamente treinam políticas representadas por árvores de decisão a partir de demonstrações de uma política especialista. Comparamos o uso de árvores de decisão tradicionais com árvores de modelos lineares, que contém modelos lineares em suas folhas.

Palavras-chave

Aprendizado de Máquina; Aprendizado Profundo; Aprendizado por Imitação; Redes Neurais; DQN; DDQN; VIPER; Árvores de Decisão; Modelos Lineares; Políticas Interpretáveis; Automação de Tarefas; Agentes Autônomos; Simulação Ambiental; Estabilidade Operacional; Generalização de Modelos.

Abstract

Botelho, Thomas; Baffa, Augusto (Advisor). **Deriving Expert Agent Policies into Interpretable Decision Trees through Imitation Learning**. Rio de Janeiro, 2024. 50p. Tese de Doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

This study presents an investigation into the application of Imitation Learning (IL) techniques for extracting structured and interpretable policies from black-box expert models. The primary focus is to analyze the feasibility and effectiveness of this approach in translating behaviors learned by deep neural networks into decision trees, which represent a set of rules that can be sequentially evaluated to reach a decision. We evaluate this methodology in three distinct simulation environments: Lunar-Lander, Taxi, and CartPole. We test the DAgger algorithm and its variant VIPER, which iteratively train policies represented by decision trees from demonstrations of an expert policy. We compare the use of traditional decision trees with linear model trees, which contain linear models in their leaves.

Keywords

Machine Learning; Deep Learning; Imitation Learning; Neural Networks; DQN; DDQN; VIPER; Decision Trees; Linear Models; Interpretable Policies; Task Automation; Autonomous Agents; Environmental Simulation; Operational Stability; Model Generalization.

1
Introdução

A Aprendizagem por Reforço (RL) é um ramo da inteligência artificial que foca no desenvolvimento de agentes capazes de tomar decisões em ambientes interativos, com base na maximização de recompensas acumuladas. Essa abordagem geralmente utiliza o conceito de Processo de Decisão Markoviano (MDP) para modelar o ambiente, onde as ações do agente têm impacto nas situações futuras (SUTTON; BARTO, 2018, Chapter 3). Uma técnica popular dentro da RL é o Q-learning, uma forma de encontrar uma política (mapeamento de estados para ações) sem a necessidade de conhecer um modelo prévio para o ambiente (SUTTON; BARTO, 2018, Chapter 6.5). Nesta linha, o agente eventualmente aprende a estimar a recompensa esperada para cada ação em cada estado, de modo a maximizar a recompensa total ao longo do tempo.

No Q-learning tradicional, o agente utiliza tabelas de decisão, chamadas tabelas Q, para armazenar os valores Q de cada par estado-ação, indicando a qualidade esperada de realizar uma ação em um estado específico (WATKINS, 1989). No entanto, essa abordagem enfrenta dificuldades quando o espaço de estados ou ações é contínuo - dependendo do nível de granularidade da forma de discretização escolhida, o custo de memória para armazenar e gerenciar esta tabela pode se tornar proibitivamente alto. Além disso, se discretização não for suficientemente granular, pode introduzir erros de aproximação que afetam a precisão dos valores Q e, consequentemente, a qualidade das decisões do agente (LAMPTON; VALASEK, 2009).

Deep Q-Networks (DQN) são uma evolução do Q-learning tradicional, incorporando redes neurais profundas (DNNs) para lidar com ambientes de aprendizado por reforço contínuos ou com grande quantidade de estados. Ao invés de usar uma tabela para armazenar e atualizar a recompensa esperada de cada ação em cada estado, a DQN aproxima a função de valor Q através de uma rede neural. Esta abordagem permite a DQN lidar com um número muito maior de estados sem necessidade de discretização, superando métodos tabulares tradicionais neste aspecto (MNIH et al., 2013).

As políticas aprendidas a partir de DQNs possuem performance muito satisfatória mas têm baixa interpretabilidade. Neste caso, consideramos interpretabilidade como a capacidade de um modelo de providenciar explicações de suas ações de forma inteligível para um humano. Idealmente, explicações podem ser formuladas no formato de regras de decisão lógica (if-then rules) e descritas em termos familiares para alguém que entende o domínio em ques-

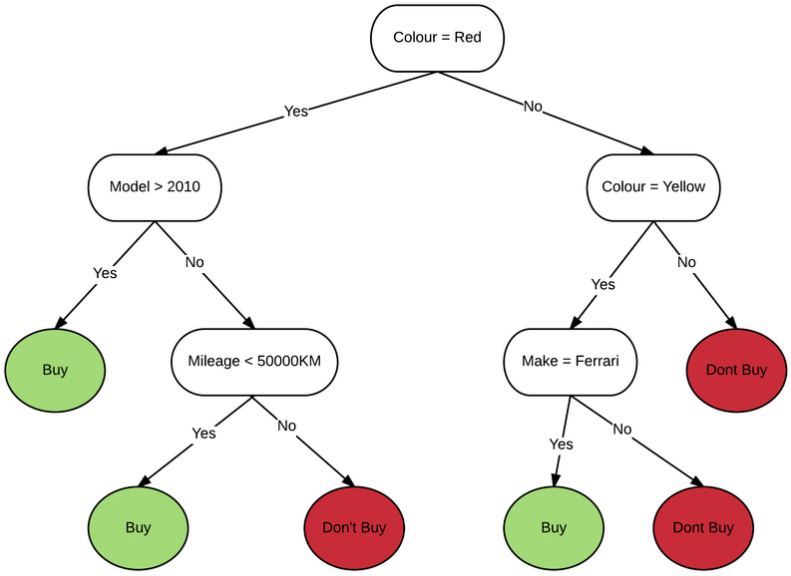


Figura 1.1: Estrutura de uma árvore de decisão binária que define regras para comprar ou não um carro (EDRAWMAX, 2024)

tão (DOSHI-VELEZ; KIM, 2017). No caso de DQNs com funções de ativação não-lineares (como ReLU ou tanh), não é claro como os pesos de diferentes nós em cada camada contribuem para uma explicação da sua predição final (KATZ et al., 2017). Quanto maior a rede, maior a complexidade resultante da densisdade das conexões entre camadas sequenciais e das interações de seus pesoas correspondentes. Isto dificulta o rastreamento do impacto de cada característica de entrada a partir de uma inspeção manual (FROSST; HINTON, 2017).

Diferente das DNNs, em uma árvore de decisão binária, a interpretabilidade é inerente à sua estrutura (FROSST; HINTON, 2017). Cada folha da árvore representa uma distribuição de probabilidade discreta contendo as classes alvo - no caso de uma política, essas são as ações. Cada antecessor de uma folha corresponde a uma condição de comparação com um atributo de entrada que, quando satisfeita, leva a uma predição ou decisão seguinte 1.1. Podemos diretamente associar as decisões sequenciais que levam uma observação de entrada a uma dada predição. Isto é representado por um caminho na árvore, da raiz até uma folha. Os nós de tal árvore formam uma hierarquia de regras de decisão lógica (BREIMAN et al., 1984, Chapter 2).

Neste trabalho buscamos comparar duas técnicas relacionadas para extrair políticas de árvore binárias a partir de políticas gerados por DNNs, o DAgger e o VIPER. Para melhorar os resultados encontrados pelos respectivos autores destas ténicas, aplicamos estas com árvores lineares binárias, que contém classificadores lineares em suas folhas. No contexto do RL, o algoritmo

DAgger (Dataset Aggregation) permite extrair representações interpretáveis de políticas a partir de demonstrações das mesmas. O DAGGER opera de forma iterativa, começando com a criação de um conjunto de dados de treinamento composto por observações de um dado agente especialista. Em seguida, treina uma árvore de decisão de forma supervisionada, para tentar aproximar o comportamento da política demonstrada. A cada iteração, o algoritmo acrescenta a base de dados para melhor capturar os comportamentos da política demonstrada. O VIPER, uma variante deste algoritmo, utiliza reamostragem para tentar melhorar este processo de refinamento da base de dados.

2
Fundamentação Teórica

Aprendizado por Reforço

Um agente treinado sob o paradigma de RL aprende a tomar decisões ótimas por meio de uma função de custo, interagindo diretamente com o ambiente. Um problema de RL é geralmente formulado como um Processo de Decisão Markoviano (SUTTON; BARTO, 2018, Chapter 3), que consiste de uma tupla de 4 elementos (S, A, P_a, R_a), sendo estes:

- o espaço de estados $S \in \mathbb{R}^n$
- o espaço de ações A
- a função que denota a probabilidade de transição de estados dado uma ação tomada em certo estado $P : S \times A \to S$
- a função de recompensa, que denota a recompensa obtida por tomar uma ação em dado estado $R : S \times A \times S \to \mathbb{R}$.

Uma política no contexto de RL é uma estratégia que o agente utiliza para determinar suas ações em um dado estado. Formalmente, uma política pode ser descrita como uma função π que mapeia estados para ações, $\pi : S \to A$. Esta função define a ação que o agente escolherá quando estiver em um determinado estado. Políticas podem ser determinísticas, onde um par de estado-ação necessariamente transiciona o agente para certo estado, ou estocásticas, onde esta transição é representada por uma distribuição de probabilidade sobre A.

Dado um Processo de Decisão Markoviano (MDP) em tempo discreto, um agente busca uma política que maximiza sua recompensa a partir de algum critério cumulativo. Tradicionalmente, para facilitar a tratabilidade do problema, consideramos esta otimização sob um horizonte de tempo infinito, assim limitando o espaço de busca à políticas estacionárias. Aplicando um fator de desconto aplicado a cada passo de tempo obtemos:

$$R_t = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{at}(s_t, \pi(s_t))\right] \tag{2-1}$$

onde $0 < \gamma < 1$ é o fator de desconto e $R(s_t, \pi(s_t))$ é a recompensa recebida ao tomar a ação $\pi(s_t)$ no estado s_t ditado pela política π (SUTTON; BARTO, 2018, Chapter 3).

O objetivo principal do agente é identificar uma política ótima, denotada por π^*, que busca maximizar o total das recompensas acumuladas, R_t. O fator

de desconto influencia a percepção de valor das recompensas futuras: quanto maior o valor de γ, mais o agente valoriza as recompensas futuras, considerando as consequências de longo prazo de suas ações atuais. Por outro lado, um γ mais baixo faz com que o agente dê mais peso às recompensas imediatas, focando em benefícios de curto prazo.

A cada passo de tempo, o agente precisa selecionar uma ação $a_t \in A$, de acordo com alguma política π com base no estado atual, com $a_t = \pi(s_t)$. A aplicação desta ação leva o estado do sistema s_t ao próximo estado $s_{t+1} = R(s_t, a_t)$. O agente também recebe uma recompensa do ambiente, r_t. Um episódio de aprendizagem consiste da repetição deste processo por uma quantidade pré-determinada de passos ou até que o sistema alcance seu estado terminal.

Tipicamente, podemos diferenciar entre configurações contínuas e episódicas para os problemas que os agentes resolvem. Nos problemas contínuos, o agente interage com o ambiente indefinidamente, sem um estado terminal definido. Já nos problemas episódicos, a interação é dividida em episódios, que terminam quando o agente alcança um estado terminal ou após um número fixo de passos. Cada tipo de configuração requer abordagens diferentes para otimizar a política do agente e maximizar a recompensa acumulada.

Os cenários contínuos e episódicos podem ser unificados adicionando um "estado absorvente", que transiciona apenas para si mesmo e retorna recompensa zero (SUTTON; BARTO, 2018, Chapter 3). Assim, todos os resultados para o cenário de horizonte de tempo infinito são válidos para o cenário episódico. Isso significa que ao introduzir um estado absorvente no qual o agente permanece indefinidamente após alcançá-lo. Esse estado absorvente atua como um ponto de término natural para o agente, garantindo que ele não receba recompensas adicionais e que suas ações futuras não influenciem mais o ambiente. Portanto, mesmo em ambientes onde os episódios têm um fim natural, os algoritmos e análises desenvolvidos para horizontes de tempo infinitos ainda se aplicam.

Os algoritmos de RL geralmente se dividem em duas categorias: os baseados em valor e os baseados em política. Nos métodos baseados em valor, a política é implícita e derivada diretamente da função de valor, que estima o retorno total esperado a partir de um estado ou estado-ação. Nos métodos baseado em política, uma representação de uma política, ou seja, um mapeamento de estados para ações, é construída diretamente durante o processo de aprendizagem (SUTTON; BARTO, 2018, Chapter 2). Para os fins deste estudo, são considerados apenas métodos baseados em valor.

Q-Learning

O manejo efetivo de recompensas atrasadas representa um desafio crítico no campo de RL. Em muitos cenários reais, as ações de um agente não resultam em recompensas imediatas, mas sim em benefícios que se manifestam após um intervalo de tempo considerável. Esta natureza de recompensas diferidas dificulta o planejamento do agente, ocluindo a associação entre suas ações e os resultados subsequentes. Em (WATKINS, 1989), o autor propõe abordagens para lidar com esse problema, estabelecendo as bases para o desenvolvimento do Q-Learning.

Um aspecto crucial no Q-Learning e em outras técnicas de aprendizado por reforço é o equilíbrio entre exploração e explotação. Exploração refere-se à busca por novas ações que podem potencialmente levar a maiores recompensas no futuro, enquanto explotação significa escolher ações que já são conhecidas por fornecer boas recompensas com base na experiência passada. O dilema exploração-explotação surge porque um agente precisa explorar suficientemente para descobrir novas estratégias eficazes, mas também precisa explotar essas estratégias para maximizar suas recompensas (SUTTON; BARTO, 2018, Chapter 2). Políticas *epsilon-greedy*, onde o agente escolhe uma ação aleatória com uma pequena probabilidade (epsilon) e a ação de maior valor Q com a probabilidade restante, são uma forma de abordar este dilema (WATKINS, 1989).

O Q-Learning é uma técnica de aprendizado por reforço que visa encontrar a política ótima de ações em um ambiente de decisão sequencial. A cada iteração, atualiza uma tabela com os valores Q de um par (estado, ação) com base nas recompensas observadas por uma política *epsilon-greedy*. Este processo representa um refinamento progressivo da política de ação. Essa abordagem permite que o agente avalie o potencial de longo prazo de suas ações, equilibrando entre a exploração de novas ações e a explotação das já conhecidas por suas altas recompensas. A natureza livre de modelo do Q-Learning torna-o particularmente poderoso em situações onde o ambiente é complexo ou desconhecido, já que o agente não precisa entender a dinâmica do ambiente completa para aprender a agir de forma eficaz.

Adicionalmente, a inicialização aleatória da tabela Q também é utilizada para promover a exploração no início do treinamento. Inicializando os valores Q de forma aleatória, o agente é incentivado a experimentar diferentes ações, já que inicialmente não possui conhecimento prévio sobre quais ações podem ser mais vantajosas. Esse processo ajuda a evitar que o agente se prenda prematuramente a uma política subótima e facilita a descoberta de estratégias

potencialmente mais eficientes ao longo do tempo.

Esse algoritmo se baseia na estimativa de uma função de valor de ação, chamada função Q, que atribui um valor para cada par ação-estado. A atualização da função Q é realizada iterativamente, utilizando a equação de Bellman de atualização do valor Q:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[R(s,a) + \gamma \max_{a'} Q(s',a') - Q(s,a) \right] \qquad (2\text{-}2)$$

Onde:

- $Q(s,a)$ é o valor de Q para o estado s e a ação a;
- α é a taxa de aprendizado;
- $R(s,a)$ é a recompensa obtida ao realizar a ação a no estado s;
- γ é o fator de desconto, determinando o peso das recompensas futuras;
- s' é o próximo estado alcançado após a realização da ação a;
- a' é a ação que maximiza o valor Q no próximo estado s'.

Em (WATKINS; DAYAN, 1992), os autores aprofundam o conceito de Q-Learning, demonstrando que uma função Q aprendida desta forma converge para os valores ótimos de ação com probabilidade 1, desde que todas as ações sejam repetidamente amostradas em todos os estados (num horizonte de tempo infinito) e os valores de ação sejam representados de forma discreta.

DQN (Deep Q-Network)

O uso de *Deep Q-Networks* (DQNs) é uma extensão do Q-Learning que integra *Deep Neural Networks* (DNNs) para estimar a função Q, permitindo lidar eficientemente com espaços de estados de grande dimensão. Sua arquitetura envolve uma rede neural, denominada rede primária, que recebe como entrada o estado atual e com isso gera valores Q para cada ação possível. Inicialmente, os valores Q alvo no DQN são estabelecidos com base nas predições da própria rede, que é aleatoriamente inicializada. Uma cópia desta rede, chamada de rede alvo, é periodicamente ajustada de acordo com um intervalo de tempo suficientemente pequeno para evitar atualizações excessivamente rápidas e instáveis (MNIH et al., 2015). A atualização dos pesos da rede primária se dá pela minimização de uma função de perda que compara valores Q previstos com os alvos.

O uso de duas redes ajuda a mitigar uma das principais dificuldades deste método, a correlação entre os valores Q previstos e os alvos, que dificulta a convergência do algoritmo. Enquanto rede primária é responsável

por selecionar as ações a serem executadas, a rede alvo é utilizada para calcular os valores Q alvo que serão usados na função de perda. A rede alvo começa como uma cópia da rede primária, mas diverge a medida que seus pesos são atualizados menos frequentemente. Em vez de serem atualizados a cada iteração do algoritmo, são sincronizados periodicamente (por exemplo, a cada 4 ou 8 iterações) com os pesos da rede primária. Esse processo de sincronização reduz a correlação entre os valores Q previstos e os alvos, promovendo maior estabilidade no treinamento (MNIH et al., 2013).

A equação de Bellman para o DQN, considerando a separação em duas redes, é dada por:

$$Q(s,a,\theta) \leftarrow Q(s,a,\theta) + \alpha \left[R(s,a) + \gamma \max_{a'} Q'(s',a',\theta') - Q(s,a,\theta) \right] \quad (2\text{-}3)$$

Onde:

- $Q(s,a,\theta)$ é o valor de Q para o estado s e a ação a na rede primária com parâmetros θ;
- $Q'(s',a',\theta')$ é o valor de Q na rede neural alvo com parâmetros θ';
- α é a taxa de aprendizado;
- $R(s,a)$ é a recompensa obtida ao realizar a ação a no estado s;
- γ é o fator de desconto, determinando o peso das recompensas futuras;
- s' é o próximo estado alcançado após a realização da ação a;
- a' é a ação que maximiza o valor Q no próximo estado s'.

O "experience replay"é uma técnica para eficientizar o processo de aprendizado de uma DQN. Neste intuito, um buffer de replay armazena experiências passadas tal que:

- $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$ é uma experiência,
- s_t é o estado atual,
- a_t é a ação tomada,
- r_{t+1} é a recompensa recebida,
- s_{t+1} é o novo estado.

Durante o treinamento, em vez de usar apenas a experiência mais recente, um conjunto aleatório dessas experiências é agregada a partir da política aprendida corrente (MNIH et al., 2013). Isso diminui a correlação entre experiências sequenciais e distribui o aprendizado mais uniformemente ao longo de diferentes estados (POMERLEAU, 1991). O buffer é atualizado continuamente, com novas experiências sendo adicionadas enquanto as antigas

são eventualmente descartadas, mantendo um equilíbrio entre experiências recentes e passadas. O intuito desta técnica é estabilizar a convergência na busca por uma política ótima.

DDQN (Double DQN)

O método DDQN é uma variação do DQN que visa mitigar a sobreestimação dos valores de ação estimados pela rede neural. No DQN tradicional, a mesma rede neural é usada para selecionar a melhor ação e estimar o valor Q dela no estado seguinte, o que tende a causar uma superestimação destes valores Q. O Double DQN foi introduzido no artigo original de 2015 (HASSELT; GUEZ; SILVER, 2015), para melhorar a estabilidade e o desempenho do algoritmo. A DDQN usa a rede primária para selecionar as ações e computar seus valores Q no estado atual, e a rede alvo para calcular os valores Q das ações selecionadas no estado seguinte. A principal motivação para essa abordagem é resolver a instabilidade no treinamento causado pela interação entre os valores Q estimados e os valores Q usados como alvo na atualização. A rede primária é atualizada a cada etapa do treinamento e estima os valores Q das ações atuais, enquanto a rede alvo é usada para calcular os valores Q das ações seguintes (HASSELT; GUEZ; SILVER, 2015).

A equação de Bellman do Double DQN é semelhante à do DQN, mas com uma pequena diferença no cálculo dos valores Q alvo:

$$Q(s,a,\theta) = Q(s,a,\theta) + \alpha \left[R(s,a) + \gamma Q'(s', \underset{a}{\text{argmax}} Q(s',a,\theta)), \theta') - Q(s,a,\theta) \right] \tag{2-4}$$

Onde:

- $Q'(s', \underset{a}{\text{argmax}} Q(s',a,\theta), \theta')$ é a estimativa dos valores Q usando a rede alvo.
- $\underset{a}{\text{argmax}} Q(s',a,\theta)$ é a ação selecionada pela rede primária com base nos valores Q estimados pela própria rede.

Imitation Learning

Muitas vezes existem necessidades relacionadas a memória, tempo de execução ou verificabilidade, que são satisfeitas com a aprendizagem de um modelo mais simples ou estruturado a partir de demonstrações de um oráculo (BUCILA; CARUANA; NICULESCU-MIZIL, 2006). Aprendizado por Imitação compreende uma variedade de técnicas para aprender uma política a

partir de demonstrações de tais demonstrações. Essas demonstrações são tradicionalmente apresentadas na forma de trajetórias, ou seja, sequências de pares estado-ação, onde cada par indica a ação a ser tomada no estado visitado. Uma das primeiras técnicas neste âmbito é conhecida como Behavioral Cloning (BC), e trata do problema como uma tarefa de aprendizado supervisionado (ROSS; BAGNELL, 2010). Na literatura, muitas vezes o modelo mais simples é denominado como aprendiz e o oráculo como especialista — adotamos aqui esta convenção quando apropriado.

Um dos primeiros usos de BC foi no treinamento de um veículo autônomo a partir dos dados de um simulador de direção em uma estrada (POMERLEAU, 1991). Neste estudo, D. Pomerleau gerou um conjunto de trajetórias a partir de uma câmera ligada em um carro dirigido por um humano. Treinou uma rede neural profunda para determinar a orientação e curvatura da estrada dado uma imagem da estrada. A rede treinada foi então validada como controlador de um carro autônomo em algumas estradas restritas. O autor comenta que como o especialista tendia a se manter no meio da estrada, a base de dados do aprendiz não era diversa o suficiente para ensiná-lo a se recuperar em situações problemáticas. Ele resolve este problema com a adição de dados sintéticos, rotacionando aleatoriamente imagens consideradas muito similares no conjunto de dados de entrada.

Começamos introduzindo uma notação relevante para o nosso cenário, de acordo com Ross e Bagnell (ROSS; GORDON; BAGNELL, 2011). Denotamos por π a classe de políticas que o aprendiz está considerando e T o horizonte de tempo para uma tarefa. Para dada política π, seja d^{π} a frequência de estados seguindo π do início até o instante t. Denominamos

$$d^{\pi}(s) = \frac{1}{T}\sum_{t=1}^{T} d_t^{\pi}(s) \tag{2-5}$$

a frequência de visita normalizada aos estados observados de $t = 1$ até $t = T$. Dado um estado s e uma ação a, seja $C(s, a) \in [0, 1]$, o custo imediato esperado de realizar a no estado s para a tarefa em consideração. O custo imediato esperado de seguir π no estado s é dado por

$$C^{\pi}(s) = \mathbb{E}_{a\sim\pi}[C(s, a)] \tag{2-6}$$

No caso onde a recompensa do nosso ambiente é conhecida, podemos considerar $C(s, a)$ como a função de recompensa negativa normalizada (tornando um problema de maximização em um problema de minimização). O custo total para executar a política durante T-etapas é

$$J^{\pi} = \sum_{t=1}^{T} \mathbb{E}_{s \sim d^{\pi}}[C^{\pi}(s)] \tag{2-7}$$

No aprendizado por imitação, tipicamente não conhecemos a verdadeira função $C(s, a)$ para nossa tarefa. Em vez disso, observamos demonstrações de um especialista e buscamos limitar a diferença $J(\pi) - J(\pi^*)$ para uma função de custo com base em quão bem π imita a política do especialista π^*. Supondo que não conseguimos computar $C(s, a)$ diretamente, usamos uma função de perda empírica $\mathcal{L}$, que minimizamos em seu lugar. Se quisermos explicitamente minimizar a quantidade de erros cometidos pelo aprendiz, podemos considerar $\mathcal{L}$ como a perda 0-1 de π com respeito a π^* no estado s,

$$\mathcal{L}(s, \pi) = \mathbb{I}[\pi(s) \neq \pi^*(s)], \tag{2-8}$$

onde $\mathbb{I}$ é a função indicadora.

Note que em casos em que queremos otimizar a capacidade do aprendiz de prever as ações escolhidas por um especialista, C e $\mathcal{L}$ podem representar a mesma função. Idealmente, queremos encontrar uma política que minimize $\mathcal{L}$ sob uma distribuição de estados induzida por uma possível política π,

$$\pi^* = \arg\min_{\pi} \mathbb{E}_{s \sim d^{\pi}}[\mathcal{L}(s, \pi)] \tag{2-9}$$

A distribuição d^{π} depende da política π escolhida, ou seja, estamos minimizando sob um conjunto de distribuições não-idênticas. Isto viola a suposição de dados independentes e identicamente distribuídos (i.i.d) que é necessária para o funcionamento de muitos algoritmos de aprendizado supervisionado (ROSS; BAGNELL, 2010).

Uma abordagem supervisionada tradicional para este problema é desconsiderar a variação entre as distribuições amostradas no treinamento, usando apenas a distribuição do especialista para recolher e rotular tais observações (decidir qual ação tomar):

$$\pi_{\text{sup}} = \arg\min_{\pi} \mathbb{E}_{s \sim d^{\pi^*}}[\mathcal{L}(s, \pi)] \tag{2-10}$$

Neste caso, a violação da suposição de i.i.d ocorre porque a distribuição dos conjuntos de treino e teste diferem. As observações amostradas durante o treinamento vêm da política do especialista enquanto as de teste, usadas para avaliar o modelo, vêm do aprendiz. O problema desta abordagem é o mesmo notado por D. Pomerleau — o aprendiz pode cometer erros que provocam trajetórias em que cada decisão incorreta aumenta a chance de futuras decisões incorretas, devido à falta de exemplos sobre como deveria se recuperar. Tais trajetórias são denominadas de degeneradas. Mais formalmente, Ross e Bagnell (ROSS; GORDON; BAGNELL, 2011) demonstram que, considerando um

aprendiz π, que comete erros com probabilidade δ, a quantidade de erros cometidos em expectativa pelo aprendiz cresce, no pior caso, quadraticamente de acordo com o horizonte de tempo T:

$$\mathbb{E}[H(\pi)] \leq J(\pi^*) + T \cdot \delta^2 \tag{2-11}$$

Este resultado vale para a função de perda 0-1 e outras funções que são limites superiores convexos desta. No pior caso, com a perda 0-1 e $\delta = 1$, onde o aprendiz comete um erro em todos os estados possíveis, temos $J(\pi^*) = 0$ e $J(\pi)$ igual a soma dos números naturais de 1 até $T - 1$:

$$S_T = \frac{T(T+1)}{2} = J(\pi^*) - J(\pi) = O(T^2\delta) \tag{2-12}$$

Para melhorar este pior caso, precisamos levar em consideração o deslocamento da distribuição amostral durante o processo de otimização, para intuitivamente evitar trajetórias degeneradas. O estudo de Ross, Gordon e Bagnell (ROSS; GORDON; BAGNELL, 2011) intitulado "A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning"propõe um algoritmo chamado DAgger (Dataset Aggregation), que garante, para certos modelos com funções de perda fortemente convexas, um limite superior melhor que a aplicação direta do aprendizado supervisionado:

$$J(\pi) \leq J(\pi^*) + O(T\sqrt{\delta}) \tag{2-13}$$

Considerando a necessidade de otimizar sobre uma distribuição de estados não estacionária, o DAgger busca iterativamente adicionar observações relevantes ao conjunto de dados. Este processo ocorre ao longo de N iterações, treinando um aprendiz a cada iteração. Na primeira iteração, começando com um conjunto de dados vazio, o especialista é solicitado a fornecer várias trajetórias de comprimento T para um determinado ambiente. Com base nessas trajetórias iniciais, o primeiro aprendiz, denominado π_1, é treinado para classificar as ações apropriadas. Nas iterações seguintes, o algoritmo coleta novas observações utilizando o aprendiz da iteração anterior, e as rotula usando a política do especialista. Estas observações rotuladas são então agregadas ao conjunto de dados. Este processo é repetido por N iterações; no final do processo, o melhor aluno é escolhido dentre os $N-1$ treinados, a partir de um processo de validação (e.g., recompensa média obtida em 50 episódios).

Em situações onde a quantidade de dados é limitada, os primeiros aprendizes tendem a ser mais aleatórios e, consequentemente, visitam estados que se tornam menos relevantes à medida que mais dados são coletados. Neste caso, pode ser benéfico utilizar uma combinação ponderada (com decaimento exponencial) da política do especialista e da política do aprendiz para gerar novas

observações. Assim, ao longo de N iterações, o DAgger tenta progressivamente melhorar a qualidade do aprendiz, providenciando uma base de dados cada vez mais relevante no que diz respeito à presença de demonstrações especialistas de como consertar falhas em trajetórias recolhidas por aprendizes.

Algoritmo 1: DAgger Algorithm for Imitation Learning

Entrada: Your input here
Saída: The resulting policy π

Initialize $D \leftarrow \emptyset$
Initialize π to any policy in IL
para $i = 1$ **até** N **faça**
    Let $\pi_i = \beta_i \pi^* + (1 - \beta_i)\pi$
    Sample T-step trajectories using π_i
    Get dataset $D_i = \{(s, \pi^*(s))\}$ of visited states by π_i and actions given by expert
    Aggregate datasets: $D \leftarrow D \cup D_i$
    Train classifier π_{i+1} on D
Return best π_i on validation

Em um artigo intitulado "Verifiable Reinforcement Learning via Policy Extraction", os autores Bastani, Pu e Solar-Lezama apresentam o Q-DAgger, uma adaptação do DAgger que prioriza estados críticos na base de observações (BASTANI; PU; SOLAR-LEZAMA, 2018). Considerando os valores Q do especialista associados a cada estado na base, são considerados críticos os estados que têm uma grande diferença entre o Q-máximo e o Q-mínimo em relação aos outros estados. Em casos onde a função Q não é diretamente observável, consideramos $\log \pi^*(s, a)$ como o valor Q(s, a). Seja A o conjunto de ações possíveis e π^* a política do especialista, quantificamos a criticidade de um estado da seguinte forma:

- Estado crítico: $Q(s, \pi^*(s)) = \min_{a \in A} Q(s, a)$
- Estado não-crítico: $Q(s, \pi^*(s)) \gg \min_{a \in A} Q(s, a)$
- Função de criticidade: $F_c(s, a) = Q(s, \pi^*(s)) - \min_{a \in A} Q(s, a)$

Este algoritmo, denominado Q-DAgger, equivale a executar o DAgger com os parâmetros $\beta_i = \mathbb{I}[i = 1]$, e uma função de perda $L_Q(s, \pi)$ que consiste da função de perda 0-1 multiplicada pela função de criticidade:

$$L_Q(s, \pi) = F_c(s, \pi) \cdot \mathbb{I}[\pi(s) \neq \pi^*(s)]$$

onde $\mathbb{I}$ é a função indicadora. A valoração de β_i escolhida indica que a política especialista é usada para recolher observações apenas na primeira iteração, usando o aprendiz treinado na iteração anterior nas iterações seguintes.

Os autores do Q-DAgger consideram uma forma de aplicar um análogo deste algoritmo para gerar uma árvore de decisão, com intuito de produzir um modelo aprendiz que seja fácil de verificar formalmente (BASTANI; PU; SOLAR-LEZAMA, 2018). Como a função de perda de uma árvore de decisão não é convexa, as garantias teóricas sobre o limite superior da quantidade de erros cometidos pelo aprendiz não se aplicam. Devido a isso, a função de perda de tal árvore não pode ser diretamente adaptada às necessidades do Q-DAgger. O algoritmo seguinte apresentado, VIPER, prioriza os estados de acordo com sua criticidade:

$$D' = \{(s, a) \in D \mid p((s, a)) \geq F_c(s) \cdot \mathbb{I}[(s, a) \in D]\}$$

A cada iteração, após a agregação das observações mais recentes na base de dados, os estados da base são reamostrados, com peso $Q(s, \pi^*) - \min_{a \in A} Q(s, a)$.

Algoritmo 2: VIPER Decision Tree Policy Extraction

Entrada: (S, A, P, R), expert policy π^*, Q-values Q^*, M, N
Saída: Best policy π from $\{\pi_1, \ldots, \pi_N\}$ on cross-validation

Initialize dataset $D \leftarrow \emptyset$
Initialize policy $\pi \leftarrow \pi^*$
para $i = 1$ **até** N **faça**
    Sample M trajectories $D_i \leftarrow \{(s, \pi^*(s)) \sim d^{\pi_{i-1}}\}$
    Aggregate dataset $D \leftarrow D \cup D_i$
    Resample dataset $D' \leftarrow \{(s, a) \sim p((s, a)) \times F_c(s) \mid (s, a) \in D\}$
    Train decision tree $\pi_i \leftarrow \text{TrainDecisionTree}(D')$
Return Best policy $\pi \in \{\pi_1, \ldots, \pi_N\}$ on cross-validation

O VIPER fornece uma alternativa interessante para melhorar a interpretabilidade e a verificabilidade das políticas aprendidas no contexto de aprendizado por reforço. Ao combinar o aprendizado supervisionado com informações provenientes do especialista, é possível gerar modelos que não apenas são mais compreensíveis, mas também potencialmente mais confiáveis e com menor custo de inferência.

2.1 Trabalhos Relacionados

Nesta seção, apresentamos uma revisão dos trabalhos relacionados a este artigo. Esses trabalhos abordam diferentes aspectos do aprendizado por reforço e técnicas associadas, fornecendo um contexto importante para o nosso trabalho.

A referência (MNIH et al., 2013) descreve o trabalho de V. Mnih et al. Os autores apresentam um modelo que revolucionou o aprendizado profundo para entradas de alta dimensionalidade. Uma rede neural convolucional foi treinada como uma variante do Q-learning; o modelo foi testado em sete jogos do Atari obtendo-se resultados impressionantes e, até mesmo, superando a performance de um especialista humano em três jogos. Suas descobertas inovadoras no uso de Deep Q-Learning são relevantes para nosso trabalho, uma vez que também aplicamos técnicas de aprendizado profundo no controle do Lunar-Lander.

A referência (HASSELT; GUEZ; SILVER, 2015) apresenta o trabalho de Hasselt, Guez e Silver. Neste artigo, os autores propõem uma variação do algoritmo DQN chamada Double DQN, que visa reduzir a sobreestimação dos valores de Q. Sua abordagem é relevante para nosso estudo, uma vez que também utilizamos o Double DQN para melhorar o desempenho do nosso agente no controle do Lunar-Lander.

Em (BASTANI; PU; SOLAR-LEZAMA, 2018), Bastani et al. abordam o problema da aplicabilidade limitada de políticas de RL geradas por redes neurais profundas, devido à incapacidade de formalmente garantir propriedades de segurança nas mesmas. Descrevem o algoritmo VIPER para treinar políticas de árvore de decisão, que podem representar políticas complexas (já que são não paramétricas) e também podem ser verificadas de forma eficiente usando técnicas existentes. O desafio é treinar políticas de árvore que atingem boas recompensas e ao mesmo tempo não sejam excessivamente profundas. Eles usam o VIPER para aprender uma política de árvore de decisão para uma variante do Atari Pong com um espaço de estados simbólicos. Formalmente demonstram, com um verificador de SMT (Satisfiability Modulo Theorems), dada certas condições iniciais este agente nunca perde o jogo (nunca deixa a bola passar de seu limite da tela).

Em (LIU et al., 2018), Liu et al. desenvolvem uma estrutura de aprendizado de imitação para aproximação de funções-Q. Introduzem Árvores U de Modelo Linear (LMUTs) para aproximar a função-Q estimada por uma rede neural. Uma LMUT é aprendida usando um novo algoritmo on-line que se adapta bem ao cenário de jogo ativo, onde o aprendiz de imitação observa uma interação contínua entre a rede neural e o ambiente. Uma avaliação empírica demonstra que um LMUT imita uma função Q substancialmente melhor do que cinco métodos de linha de base, inclusive árvores de classificação tradicionais.

A referência (LEVINE et al., 2020) faz um levantamento de técnicas de aprendizado por reforço offline (RLO), uma abordagem que se baseia em grandes quantias de demonstrações armazenadas, em vez de interagir conti-

nuamente com o ambiente. Essa técnica permite o treinamento de agentes em ambientes onde a coleta de novos dados é perigosa ou custosa demais. Levine destaca que o RLO a necessidade de lidar com dados fora da distribuição e a exploração limitada como as maiores dificuldades desta abordagem. O autor propõe métodos que combinam restrições baseadas em políticas e regularização de valor, para tentar amenizar estes problemas, buscando eficiência no aprendizado a partir de dados estáticos. Essas técnicas são relevantes para nosso trabalho, uma vez que tratam de um extremo onde nenhum tipo de interação direta com o ambiente é realizada.

3
Método proposto

Na presente seção, focamos em delinear nossas decisões no desenvolvimento de agentes interpretáveis para certos ambientes do Farama Gymnasium, anteriormente conhecido como OpenAI Gym (TOWERS et al., 2023). Esta biblioteca de Python oferece uma variedade de ambientes simulados para experimentos em RL, com uma interface programática padronizada. Queremos treinar agentes que "resolvam"de forma eficiente estes ambientes, delineando a sequência de decisões que levam a cada ação. A métrica de sucesso, ou seja, a pontuação necessária para "resolver"cada ambiente, baseia-se nas benchmarks estabelecidas pelo próprio Gymnasium, refletindo os padrões de desempenho esperados em cada cenário específico. Como foi previamente descrito, um agente especialista será representado por uma DQN. As demonstrações deste agente serão usadas para treinar diferentes tipos de árvores de decisão, que serão os aprendizes.

3.1
Descrição dos Ambientes

Os ambientes selecionados para aplicação dessas técnicas são episódicos, ou seja, cada sessão de interação do agente com o ambiente é limitada por um conjunto específico de condições de início e término. Estas condições podem variar aleatoriamente de um episódio para o outro, ou seja, não necessariamente se repetem. Em todos esses cenários, o objetivo do agente é adquirir o máximo de recompensas possível, idealmente agindo a partir de uma política que maximiza a expectativa de recompensas sob um horizonte de tempo infinito. Visando avaliar a robustez das políticas encontradas, usamos versões "barulhentas"de dois dos ambientes para testar os agentes treinados, introduzindo vento e turbulência quando apropriado.

Outro aspecto considerado foi o critério de término do episódio para cada ambiente. Além de uma recompensa máxima para cada ambiente, utilizamos os sinais *done* e *truncated* fornecidos pela biblioteca Gymnasium, que indicam se um episódio terminou naturalmente ou se deveria terminar devido a alguma razão técnica. No LunarLander por exemplo, representam se a nave pousou ou saiu da tela, respectivamente. Inicialmente não levamos em conta situações *truncated*, mas devido ao longo tempo de treinamento observado, incorporamos essa condição ao nosso critério de término. Esta adição ajudou a reduzir significativamente o tempo de treinamento, evitando que os agentes continuassem

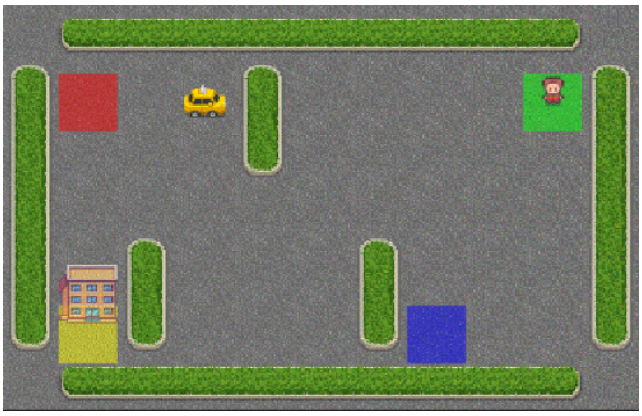


Figura 3.1: Taxi (TOWERS et al., 2023)

suas ações em episódios que claramente não seriam bem-sucedidos.

Considerando um ambiente composto de observações com c variáveis contínuas, d variáveis booleanas e um fator de discretização n, a quantidade de estados num esquema de discretização simples uniforme cresce de acordo com a expressão

$$O(n^c + 2^d)$$

Se alguns componentes das observações forem categóricos, usamos one-hot-encoding para transformá-los em atributos binários. Este crescimento exponencial ilustra a impraticabilidade dos métodos tabulares para tarefas com muitos estados, ressaltando a necessidade de abordagens mais sofisticadas que possam lidar com tal crescimento exponencial de forma eficiente. Em casos de observações acerca de quantidades físicas como velocidade angular e aceleração, pode ser muito custoso aplicar uma discretização granular o suficiente para capturar a dinâmica do ambiente.

Para cada ambiente, mostramos a quantidade de estados para cada ambiente com fatores de discretização $n = 10$ e $n = 100$. Para computar o espaço de memória ocupado por uma tabela que precisa guardar um valor para cada estado, consideramos valores de ponto flutuante IEEE 754 de 4 bytes. Para converter em gigabytes (GB), consideramos 2^{30} bytes em um GB.

Taxi

O desafio do *Taxi* consiste em um táxi que navega por uma grade 5x5 com a tarefa de buscar um passageiro em um local específico e levá-lo até seu destino. O estado do ambiente é caracterizado pela posição do táxi, a localização do passageiro e o ponto de destino. O agente tem à disposição seis

ações possíveis: mover-se para o norte, sul, leste ou oeste, pegar o passageiro e deixar o passageiro.

Vetor de Observação (Taxi):	
taxi_row	inteiro, $x \in [0, 5)$
taxi_col	inteiro, $x \in [0, 5)$
passenger_location	categórica, 5 categorias
destination	categórica, 4 categorias

O agente recebe uma recompensa de +20 ao deixar o passageiro no destino correto. Uma penalidade de -1 é aplicada a cada movimento, incentivando o táxi a completar a viagem no menor número de passos possível. Ações ilegais, como tentar pegar ou deixar o passageiro em locais inadequados, resultam em uma penalidade de -10.

O objetivo principal é desenvolver uma rota com o mínimo de movimentos desnecessários. O episódio termina quando o passageiro é entregue ao destino correto. O ambiente "Taxi-v3"no Farama Gymnasium é utilizado para simular este cenário.

Tabela 3.1: Qtd. estados Taxi (neste caso todos os componentes da observação são discretos, apenas contamos os estados))

Fator de Discretização (n)	**Qtd. Estados**	**Armazenamento (GB)**
-	$5 \times 4 \times 2^9$	$\approx 0,000038$

Cart Pole

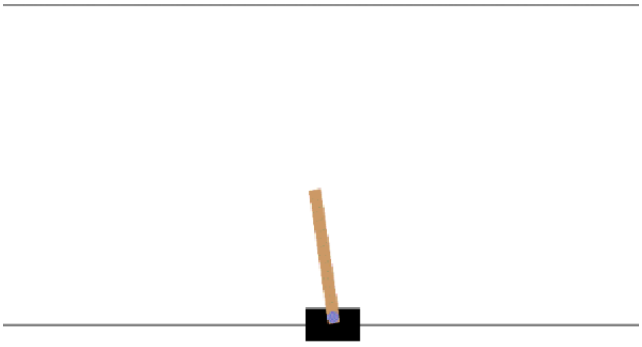


Figura 3.2: Cartpole (TOWERS et al., 2023)

O problema do *Cart Pole* é um benchmark clássico no campo da aprendizagem por reforço, que consiste em equilibrar um poste que está preso a um

carrinho que se move ao longo de um eixo horizontal. O agente tem a capacidade de aplicar forças para a esquerda ou para a direita no carrinho, visando manter o poste em posição vertical. O estado do ambiente é descrito por um vetor contendo quatro valores: a posição do carrinho, a velocidade do carrinho, o ângulo do poste em relação à vertical e a velocidade angular do poste.

Uma variante interessante deste problema foi desenvolvida para avaliar a robustez dos agentes de aprendizagem por reforço sob condições ambientais variáveis, como a presença de vento e turbulência. Nesta versão, simula-se a influência do vento através de uma força lateral que varia dinamicamente ao longo do tempo, afetando o movimento do carrinho e a estabilidade do poste. A força do vento é modelada pela função matemática:

$$f(t) = \tanh(\sin(2k(t + C)) + \sin(\pi k(t + C)))$$

de acordo com a documentação do ambiente Lunar Lander [ref], onde k é fixado em 0.01 e C é uma constante aleatória escolhida no intervalo entre -9999 e 9999 a cada episódio de treinamento. A turbulência é introduzida como uma perturbação na velocidade angular do poste, aplicada em cada instante de tempo, aumentando a dificuldade de equilibrar o poste de forma imprevisível. Esta adição de forças external ao ambiente testam a capacidade dos agentes de se adaptarem a situações que não foram vistas durante seu treinamento.

Vetor de Observação (CartPole):		
posição do carrinho	contínua	$(-4.8, 4.8)$
velocidade do carrinho	contínua	$(-\infty, \infty)$
ângulo do poste	contínua	$(-0.418 \text{ rad}, 0.418 \text{ rad})$
velocidade angular do poste	contínua	$(-\infty, \infty)$

O agente recebe uma recompensa de +1 para cada passo de tempo em que o poste permanece dentro de um certo limite angular da posição vertical e o carrinho permanece dentro dos limites definidos da pista. O objetivo é maximizar a recompensa total, mantendo o poste equilibrado pelo maior tempo possível. O ambiente "CartPole-v1"do Farama Gymnasium é utilizado para simular este cenário.

Tabela 3.2: Qtd. estados após discretização

Fator de Discretização (n)	**Qtd. Estados**	**Armazenamento (GB)**
10	10^4	$\approx 0,0004$
100	100^4	≈ 400

LunarLander

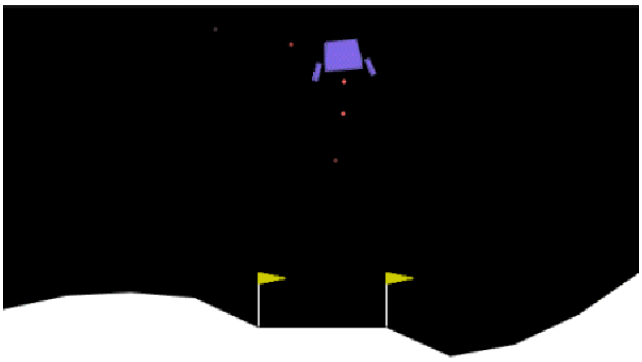


Figura 3.3: LunarLander (TOWERS et al., 2023)

O problema do *Lunar-Lander*, o mais desafiador dos escolhidos, requer o controle de uma nave espacial para um pouso seguro, manobrando entre forças gravitacionais e inércias variáveis. Consiste em controlar uma nave para pousar entre as bandeiras em uma plataforma de pouso. A nave pode executar quatro ações discretas: 0 (fazer nada), 1 (acionar motor esquerdo), 2 (acionar motor principal) e 3 (acionar motor direito). O objetivo é aprender uma política de controle que maximize a recompensa total obtida ao longo do tempo.

O estado do ambiente é representado por um vetor de dimensão 8, que inclui informações sobre as coordenadas da nave (x e y), velocidades lineares nas direções x e y, ângulo da nave, velocidade angular e o estado de contato das pernas da nave com o solo (representado por dois valores booleanos, um para cada perna da nave).

Para avaliar a robustez das políticas de controle testadas neste ambiente, foram utilizados os parâmetros de vento e turbulência previamente descritos na seção do *CartPole*. Neste caso não foi preciso modificar o código do ambiente, pois a implementação do Farama Gymnasium já providencia estes. Essas perturbações são modeladas como forças laterais e rotacionais que variam a cada instante de tempo, representando desafios adicionais para manutenção da estabilidade da nave. Estas forças externas são projetadas para testar a capacidade do agente de adaptar suas ações em resposta a mudanças rápidas e imprevisíveis.

Vetor de Observação (LunarLander):

posição x	contínua	$(-\infty, \infty)$
posição y	contínua	$(-\infty, \infty)$
velocidade x	contínua	$(-\infty, \infty)$
velocidade y	contínua	$(-\infty, \infty)$
ângulo	contínua	$[-\pi, \pi]$
velocidade angular	contínua	$(-\infty, \infty)$
perna esquerda em contato	binária	$\{0, 1\}$
perna direita em contato	binária	$\{0, 1\}$

A recompensa é determinada por várias características do estado atual da nave. Aproximar-se da plataforma de pouso resulta em uma recompensa maior, enquanto se afastar dela resulta em uma recompensa negativa. A velocidade do módulo de pouso também influencia a recompensa, com uma velocidade mais lenta sendo recompensada e uma velocidade mais rápida sendo penalizada. Além disso, a inclinação da nave afeta a recompensa, com uma nave inclinada recebendo uma penalidade. Cada perna da nave em contato com o solo fornece uma recompensa positiva adicional. Acionar os motores laterais resulta em uma penalidade menor, enquanto acionar o motor principal resulta em uma penalidade maior. Por fim, pousar com segurança na plataforma de pouso resulta em uma grande recompensa positiva, enquanto bater no solo resulta em uma grande penalidade negativa.

O estado inicial do ambiente é definido com a nave posicionada no centro superior da janela de visualização e uma força inicial aleatória aplicada ao seu centro de massa. O problema do *Lunar-Lander* é implementado utilizando a biblioteca OpenAI Gym e o ambiente "LunarLander-v2". A plataforma de pouso está sempre localizada nas coordenadas (0, 0), permitindo aterrissagens tanto na plataforma quanto fora dela. O combustível da nave é ilimitado, permitindo que o agente aprenda a voar e pousar sem restrições de combustível.

O episódio termina se a nave colidir com o solo ou se afastar da janela de visualização, onde a coordenada é maior que 1.

Tabela 3.3: Qtd. estados após discretização

Fator de Discretização (n)	**Qtd. Estados**	**Armazenamento (GB)**
10	$10^6 + 2^2$	$\approx 0,000038$
100	$10^{12} + 2^2$	$\approx 3725,29$

Metodologia

Para nosso treinar nosso modelo especialista, inicialmente recorremos ao DQN com "experience replay", que demonstrou grande instabilidade no treinamento para as tarefas escolhidas. No processo de ajuste dos hiperparâmetros, demonstrou convergência inconsistente, frequentemente não atingindo a pontuação necessária em cada ambiente. Para aprimorar a acurácia e estabilidade das estimativas de valor de ação, evoluímos para o uso da Double DQN (DDQN). Isto tornou o processo de trainamento mais estável – do ponto de vista de implementação, tivemos que acrescentar apenas uma linha de código ao projeto para fazer esta mudança.

Em seguida, implementamos o Double DQN para tentar estabilizar o processo de treinamento. Implementamos duas redes neurais: uma para selecionar as ações e outra para avaliar seus valores. Essa abordagem permitiu obter estimativas mais precisas e estáveis dos valores de ação, resultando em um processo de aprendizado mais estávek. Observamos raramente casos em que a DDQN não convergia para uma política aceitável — algo que ocorreu algumas vezes com a DQN durante o treinamento.

A topologia das redes neurais utilizadas consiste de uma camada de entrada de largura 8 representando os estados, duas camadas ocultas com 64 neurônios, e uma camada de saída com quatro neurônios representando as ações disponíveis. Durante o processo de treinamento do agente, fixamos os seguintes hiperparâmetros:

- No passo de atualização dos valores-Q, usamos um fator de desconto (gamma) = 0.99.
- Para o replay buffer, utilizamos um buffer com 20.000 posições e um "batch size"de tamanho 128.
- Para otimização de pesos durante o treinamento, foi escolhido o otimizador ADAM, com taxa de aprendizado (learning rate, LR) = 1e-3.
- Para transferência de pesos entre a rede alvo e a rede local utilizamos um soft update com taxa de transferência (tau) = 1e-3. Esta transferência ocorre a cada 4 passos de treinamento da rede alvo (update_every).

Os valores destes hiperparâmetros foram determinados com um procedimento de grid search, com as seguintes possibilidades:

- gamma = [0.99, 0.98, 0.97, 0.96, 0.95]
- batch_size = [256, 128, 64, 32, 16]
- LR =[1e-2, 1e-3, 5e-4, 1e-4, 5e-5]

- tau=[1e-2, 1e-3, 5e-4, 1e-4, 5e-5]
- update_every = [16, 8, 4, 2, 1]

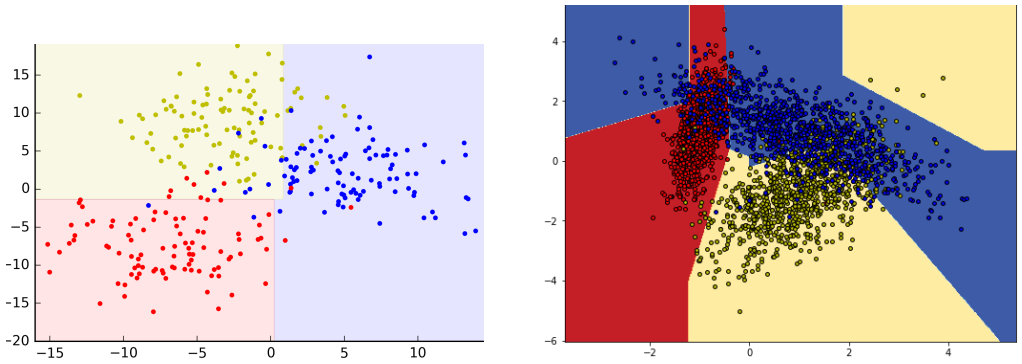
Utilizamos a estratégia *epsilon-greedy* de *exploration-exploitation* com ajuste gradual do parâmetro ϵ por um fator de decaimento constante de 0.995 em nossos experimentos. A fim de equilibrar adequadamente entre exploração e explotação ao longo do tempo, o valor de ϵ é ajustado gradualmente ao longo dos episódios de treinamento. Em nossos experimentos, utilizamos um fator de decaimento constante de 0.995 para o parâmetro ϵ, o que significa que após cada episódio, o valor de ϵ é multiplicado por 0.995. Esse processo de decaimento garante que o agente comece com uma tendência maior para explorar o ambiente (com um ϵ inicialmente alto) e, gradualmente, conforme ganha experiência, passe a confiar mais nas estimativas de valor Q para tomar decisões, reduzindo a probabilidade de escolher ações aleatórias. Isso equilibra a necessidade de descobrir novas estratégias com a necessidade de otimizar as estratégias existentes no processo de busca.

Após uma análise comparativa das curvas de validação para Deep Q-Network (DQN) e Double Deep Q-Network (DDQN), optamos pelo DDQN como o modelo especialista devido ao seu desempenho superior. O modelo especialista foi utilizado para coletar trajetórias demonstrativas, que por sua vez foram usadas para treinar três modelos diferentes de árvore de decisão (aprendizes), de acordo com o algoritmo VIPER. Os modelos utilizados nesta etapa incluem:

1. Árvore de Classificação Tradicional: Este modelo segue a abordagem convencional de árvore de decisão, criando um classificador cuja lógica se baseia em estimar a frequência de uma classe em subconjuntos dos dados de treinamento. Ele divide o espaço de características em regiões distintas por meio de cortes alinhados aos eixos, atribuindo a cada uma delas a classe mais frequente entre as amostras que caem nessa região.

2. Árvore de Classificação Linear com um Classificador Ridge: Este modelo aprimora a abordagem tradicional ao incorporar classificadores lineares nos nós. Usando um classificador Ridge, que inclui regularização L2 dos coeficientes, o modelo cria uma fronteira de decisão linear. Para classificação, transforma os alvos em valores no intervalo $[-1, 1]$ e utiliza a regularização L2 para penalizar coeficientes excessivamente grandes, prevenindo o sobreajuste e visando melhorar a generalidade do modelo. Isso permite capturar relações lineares entre as características de maneira mais robusta do que cortes puramente alinhados aos eixos.

3. Árvore de Classificação Linear com Regressão Logística: Este modelo também incorpora classificadores lineares nos nós, utilizando regressão logística para medir a perda. Com este tipo de regressão, o modelo pode estimar probabilidades de classe, o que melhora a capacidade de lidar com situações em que as classes não são perfeitamente separáveis, oferecendo uma abordagem mais robusta e flexível.



(a) Árvore binária tradicional (b) Árvore linear

Figura 3.4: Fronteiras de decisão de diferentes tipos de árvore: (a) fronteira de decisão de uma árvore linear em um *dataset* de classificação sintético com 2 atributos de entrada (normalmente distribuídos) e 3 classes (b) fronteira de decisão de uma árvore tradicional em um *dataset* com as mesmas características do anterior, mas com uma distribuição normal deslocada e com desvio padrão maior. (CERLIANI, 2024)

Para fins demonstrativos, a Fig 3.5 contém uma árvore de decisão treinada para o Lunar Lander com apenas 3 níveis de profundidade. Cada caixa colorida no diagrama abaixo corresponde a um nó da árvore. A primeira linha de texto em cada caixa contém a regra de decisão utilizada no nó correspondente (por exemplo, $\theta \leq -0.002$). A última linha corresponde à predição que seria tomada por esta folha de acordo com a distribuição de frequência naquela folha.

Todas as árvores foram induzidas a partir do algoritmo guloso CART (BREIMAN et al., 1984), construídas de forma iterativa: em cada passo, a divisão que proporciona a maior melhoria imediata em dado critério estatístico é escolhida como a decisão do nó corrente. Uma divisão consiste em comparar dividir um conjunto de pontos em dois subconjuntos, pontos para maiores que e menores que o ponto de corte. Isso é feito sem levar em conta se essa escolha levará à melhor solução considerando as decisões dos nós seguintes. Embora esse método possa não sempre resultar na árvore ótima, ele tem a vantagem de ser computacionalmente eficiente e prático para grandes conjuntos de dados.

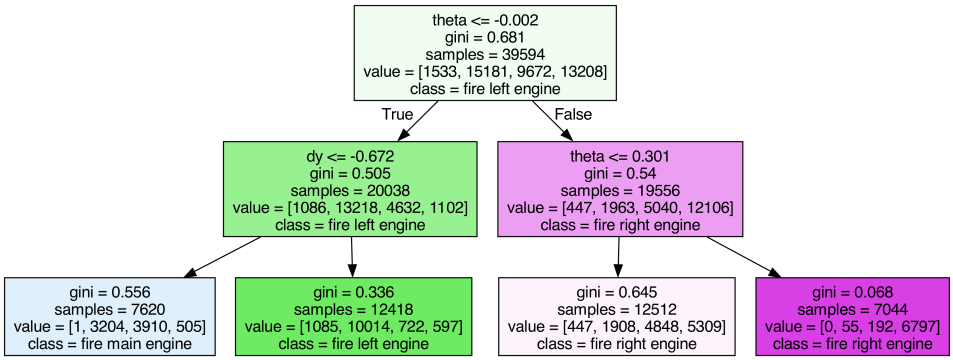


Figura 3.5

Como estamos lidando com um problema de classificação multi-classes, o coeficiente de Gini é usado para avaliar a eficácia de uma regra de divisão na separação dos dados em classes distintas. Este coeficiente mede a probabilidade de um item selecionado aleatoriamente ser classificado incorretamente se for rotulado aleatoriamente de acordo com a distribuição das classes no nó. considerando p_i como a proporção de exemplos da classe i em dado nó, temos a formula

$$Gini = 1 - \sum(p_i^2)$$

Executamos o processo com os 3 tipos de árvores de decisão. Utilizamos os seguintes parâmetros para o algoritmo VIPER, notando que quando o parâmetro *is_reweight* for verdadeiro, este algoritmo equivale ao DAgger:

- `n_batch_rollouts = 100`: Define o número de trajetórias de episódios a serem agregadas a base de dados a cada iteração de treinamento.
- `max_samples = 500_000`: Estabelece o número máximo de amostras a serem guardadas na base de dados.
- `max_iters = 50`: Determina o número de iterações no processo de treinamento, este é o nosso critério de parada.
- `train_frac = 0.8`: Especifica a fração dos dados a serem utilizados para treinamento (80%), ou outros 20% sao usados para validação.
- `n_test_rollouts = 100`: Define o número de episódios a serem executados para validar a quantia de recompensas obtidas por uma política.
- `max_depth`: Especifica a profundidade máxima das árvores de decisão treinadas, com os valores testados variando entre 1, 2, 4, 6, 8.
- `is_reweight = True`: Indica se as amostras devem ser reponderadas de acordo com sua criticidade durante o treinamento.

Esses parâmetros foram escolhidos devido a restrições de eficiência, optando-se pela maior precisão possível dentro de um limite de tempo de execução de 6 horas por ambiente (considerando todas as profundidades testadas).

Na aplicação do algoritmo VIPER, a DDQN atua como o especialista, fornecendo uma política caixa-preta de alta performance, enquanto uma família de árvores de decisão, treinadas com CART em diferentes iterações da base de dados, atuam como alunos. Ao final do processo de aprendizado, os alunos competem entre-si, resultando no modelo com a melhor performance no cenário em questão. Esta árvore de decisão, além de herdar a eficácia do especialista, demonstra seu raciocínio em regras de decisão lógicas sequenciais.

4
Resultados

4.1
DQN e DDQN

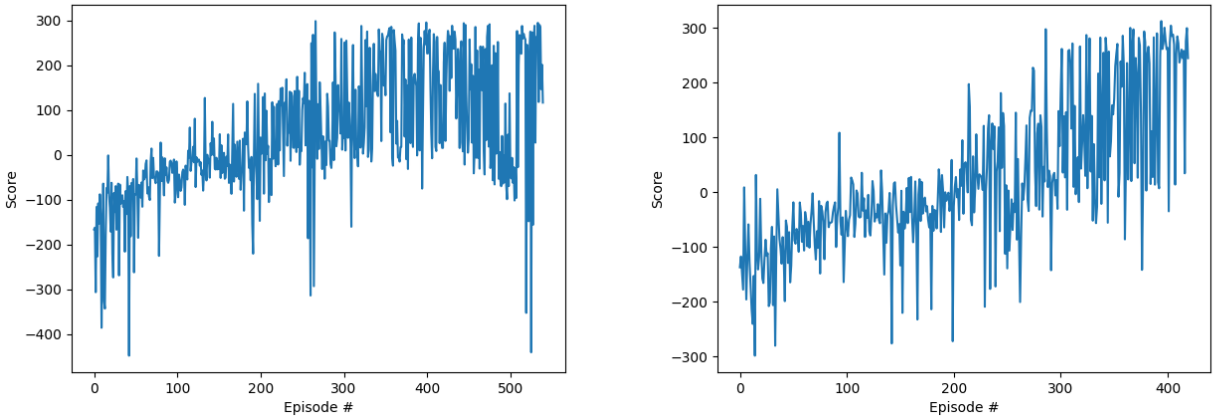
Para escolher o modelo que representa o agente especialista, repetimos 5 vezes o processo de treinamento no ambiente LunarLander, para DQN e DDQN. Se o processo demorar mais de 5k iterações para convergir, paramos a otimização e consideramos que o agente falhou em sua tarefa de resolver o ambiente. Escolhemos, para cada modelo, o resultado que convergiu em menos iterações, com intuito de analisar a curva de aprendizado dos potenciais especialistas ao longo do treinamento.

O processo de treinamento da DQN falhou em 3 das 5 tentativas, convergindo em 540 iterações no melhor caso. No caso da DDQN, convergiu para uma política aceitável em todas as tentativas, convergindo em 420 iterações. Em geral, observamos uma melhoria gradual no desempenho dos dois modelos ao longo dos episódios, com a DQN apresentando mais instabilidade, caracterizada por episódios repentinos com recompensas negativas próximos ao final do treinamento. No melhor caso, os dois agentes efetivamente aprenderam a controlar o pouso da nave no centro da fase, como se pode observar nos gráficos de treinamento da Fig 4.1.

Fica claro pelos gráficos que ambos agentes apresentaram episódios em que a recompensa atingiu valores negativos após atingir valores próximos do máximo — isto pode ser explicado pela característica "exploration-exploitation"do processo de treinamento. Durante a exploração, os agentes executam ações aleatórias para tentar descobrir estratégias potencialmente melhores, temporariamente os levando a resultados subótimos. Este modo de escolher ações durante o treinamento evita que os agentes fiquem presos em soluções localmente ótimas e os ajuda a maximizar a recompensa no longo prazo.

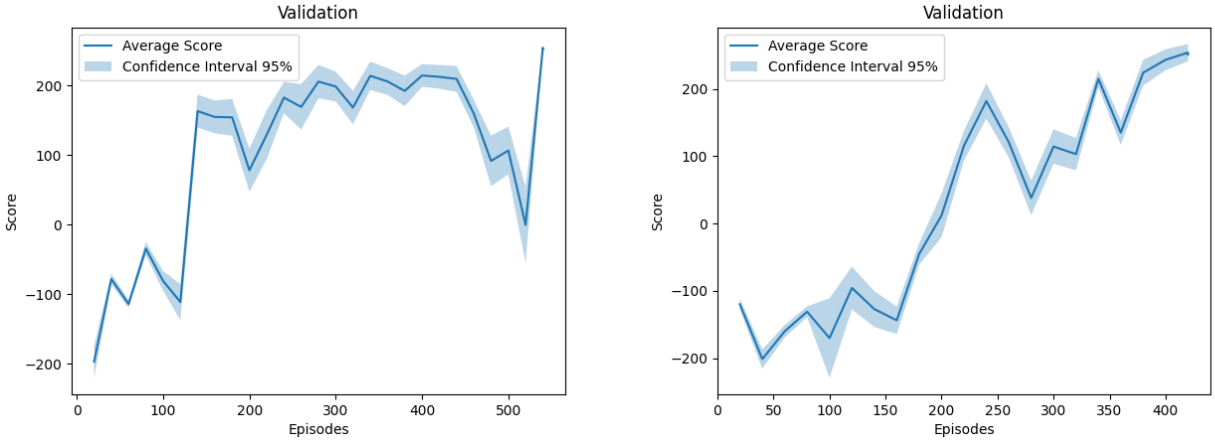
Além disso, a exploração é um componente crucial para a capacidade dos agentes de adaptar-se a mudanças e aprender novas estratégias. Na prática, a exploração pode causar flutuações no desempenho à medida que o agente experimenta novas ações, algumas das quais podem ser menos eficientes. Idealmente, a frequência de episódios com pontuações ruins deve diminuir à medida que o agente aprende a corrigir os erros induzidos pela aleatoriedade. Contudo, é importante monitorar e ajustar a taxa de exploração para garantir

que o agente não permaneça em um estado de exploração excessiva, o que pode impedir a convergência para uma política estável e ótima.



(a) DQN (b) DDQN

Figura 4.1: Curvas de pontuação média ao longo de 100 episódios (Lunar Lander)



(a) DQN (b) DDQN

Figura 4.2: Curvas de validação de 100 episódios a cada 20 de treinamento (Lunar Lander)

A Fig 4.2 mostra as curvas de validação dos agentes. Avaliamos a recompensa obtida por dado agente em blocos de 100 episódios a cada 10 episódios de treinamento. A envoltória destes gráficos representa um intervalo de confiança de 95% obtida assumindo uma distribuição **t** para cada bloco de 100 episódios. Esses resultados fornecem uma visão geral do progresso do agente ao longo do treinamento, com a DDQN melhorando de forma gradual versus os altos e baixos da DQN, principalmente ao final do treinamento.

Com base na estabilidade superior do processo de treinamento do DDQN e sua convergência em menos iterações, adotamos este agente como o especialista no processo de Imitation Learning.

4.2 Imitation Learning - VIPER

Treinamos nossos agentes aprendizes a partir do algoritmo VIPER, com intuito de extrair de políticas de imitação a partir de um modelo especialista. Neste contexto, avaliamos o desempenho de três tipos de árvores de decisão: a árvore de classificação tradicional, a árvore linear com um classificador Ridge e a árvore linear com regressão logística.

Para cada tipo de árvore, realizamos testes de 1k episódios de cada ambiente com profundidades variando entre 1, 2, 4, 6 e 8. A profundidade da árvore de certa forma determina a complexidade do modelo e sua capacidade de generalização. Profundidades maiores permitem capturar padrões mais complexos nos dados, enquanto profundidades menores podem resultar em modelos mais simples e generalizáveis, mas potencialmente menos precisos.

Ambientes Padrões As tabelas abaixo (Tab. 4.1, Tab. 4.2 e Tab. 4.3) apresentam a seleção das duas primeiras árvores de decisão para cada combinação de ambiente e tipo de modelo, ordenadas por recompensa média e desempate por tempo de execução. Esses resultados nos permitem comparar a eficácia de diferentes configurações de árvores de decisão na replicação do comportamento dos agentes especialistas treinados nos ambientes CartPole-v1, LunarLander-v2 e Taxi-v3. Os dados completos se encontram na seção de Resultados Adicionais (Cap. 6), nas tabelas Tab. 6.1, .Tab. 6.2 e Tab. 6.3.

Tabela 4.1: Top 2 - CartPole

model_type	max_depth	is_reweight	avg_reward	avg_time
DDQN	NaN	NaN	500.000000	42.808232
linear_tree_logistic	2	True	500.000	19.299410
linear_tree_logistic	2	False	500.000	24.434292
linear_tree_ridge	1	True	500.000	33.566142
linear_tree_ridge	2	False	500.000	38.832968
decision_tree	4	False	469.256	26.632576
decision_tree	8	False	448.221	26.254955

Tabela 4.2: Top 2 - LunarLander

model_type	max_depth	is_reweight	avg_reward	avg_time
DDQN	NaN	NaN	257.970973	48.717298
linear_tree_logistic	4	True	252.330563	50.923496
linear_tree_logistic	6	True	228.858184	58.217474
linear_tree_ridge	6	True	225.913827	82.674857
linear_tree_ridge	1	False	203.068736	38.130497
decision_tree	8	True	44.775291	30.754009
decision_tree	8	False	29.449678	30.590557

Tabela 4.3: Top 2 - Taxi

model_type	max_depth	is_reweight	avg_reward	avg_time
DDQN	NaN	NaN	7.940000	48.295859
linear_tree_logistic	4	True	8.002	34.873965
linear_tree_logistic	8	True	7.945	41.724709
linear_tree_ridge	4	False	5.558	56.763811
linear_tree_ridge	4	True	0.022	70.470901
decision_tree	8	True	-95.010	27.526508
decision_tree	8	False	-101.157	27.339854

A análise dos resultados mostra que, no ambiente CartPole, o modelo DDQN e as árvores lineares alcançaram a recompensa máxima de 500 em diferentes configurações. As árvores lineares, especialmente com profundidades 2 e 1, destacaram-se por sua eficiência, com tempos de execução melhor que a rede neural. As árvores de decisão tradicionais também foram eficazes, mas com recompensas ligeiramente inferiores e tempos de execução maiores que seus competidores lineares de profundidade menor.

No ambiente LunarLander, o DDQN novamente apresentou a maior recompensa média. As árvores lineares, principalmente as de regressão logística (com profundidades de 4 e 6), mostraram o melhor desempenho dos aprendizes, chegando aproximadamente entre 5% e 10% da performance do especialista. As árvores de decisão tradicionais, mesmo na maior profundidade, tiveram desempenho insatisfatória, indicando limitações na sua capacidade expressiva para escolher as.

No ambiente Taxi, as árvores lineares com regressão logística apresentaram o melhor desempenho, com as recompensas médias e tempos de execução. O DDQN apresentou recompensas ligeiramente inferiores, com tempo de execução 25% pior. As árvores de decisão tradicionais apresentaram recompensas muito inferiores, indicando a necessidade de maior profundidade para modelar este ambiente.

Ambientes Robustos Para avaliar a robustez dos aprendizes obtidos, testamos os mesmos em versões robustas do CartPole e LunarLander, que incluem vento e turbulência para simular condições mais realistas. No entanto, não há uma versão robusta do ambiente Taxi, pois não encontramos uma forma justa de adicionar ruído neste ambiente discreto sem fundamentalmente alterar suas regras. As tabelas abaixo mostram os resultados dos dois melhores modelos para esses ambientes robustos, destacando a capacidade de alguns modelos de manter um desempenho consistente em condições adversas.

As Tabelas 4.4 4.5 apresentam o melhor modelo para de cada tipo para as versões robustas do CartPole e LunarLander, respectivamente. Buscamos escolher o maior valor de turbulência que ainda permitisse o agente a progredir significativamente em direção ao objetivo, sem ficar preso se corrigindo excessivamente. Os valores para força de vento variam de acordo com um incremento fixo. As diferenças observadas entre as recompensas médias entre os ambientes padronizados e suas versões turbulentes proporcionam *insights* interessantes sobre a robustez das políticas aprendidas. Os resultados completos destes experimentos se encontram no apêndice.

Tabela 4.4: CartPole - Top 1 - Recompensas médias de 1k episódios com vento e turbulência
(enable_wind=True, turbulence_power = 0.1)

model_type	max_depth	reweight	avg_reward	avg_time	turbulence_power	wind_power
linear_tree_ridge	1.0	True	315.585	56.564687	0.1	0.1
linear_tree_logistic	2.0	True	298.251	45.959514	0.1	0.1
DDQN	NaN	NaN	296.794	32.202289	0.1	0.1
decision_tree	6.0	False	227.880	24.657906	0.1	0.1
linear_tree_logistic	1.0	False	173.521	39.763403	0.1	0.2
DDQN	NaN	NaN	163.474	32.738032	0.1	0.2
linear_tree_ridge	1.0	True	161.135	61.299682	0.1	0.2
decision_tree	8.0	False	152.502	25.124091	0.1	0.2
linear_tree_logistic	1.0	False	152.362	38.198518	0.1	0.3
DDQN	NaN	NaN	145.281	33.225942	0.1	0.3
decision_tree	6.0	True	128.894	25.544641	0.1	0.3
linear_tree_ridge	6.0	False	126.940	104.131813	0.1	0.3
linear_tree_logistic	1.0	False	124.863	38.776649	0.1	0.4
DDQN	NaN	NaN	117.432	33.431244	0.1	0.4
linear_tree_ridge	4.0	False	98.959	81.857060	0.1	0.4
decision_tree	8.0	False	97.861	26.085891	0.1	0.4
DDQN	NaN	NaN	95.036	33.473905	0.1	0.5
linear_tree_logistic	1.0	False	88.788	38.387450	0.1	0.5
decision_tree	6.0	True	79.516	25.960644	0.1	0.5
linear_tree_ridge	6.0	False	74.860	103.359457	0.1	0.5

Tabela 4.5: Lunar Lander - Top 1 - Recompensas médias de 1k episódios com vento e turbulência
(enable_wind=True, turbulence_power = 2.0)

model_type	max_depth	reweight	avg_reward	avg_time	turbulence_power	wind_power
linear_tree_logistic	6.0	True	159.699392	75.211113	2.0	0.0
DDQN	NaN	NaN	138.887504	51.795167	2.0	0.0
linear_tree_ridge	6.0	True	115.281745	103.952109	2.0	0.0
decision_tree	8.0	True	-8.431982	34.796988	2.0	0.0
linear_tree_logistic	1.0	False	139.191163	18.775421	2.0	5.0
DDQN	NaN	NaN	119.409141	51.027988	2.0	5.0
linear_tree_ridge	1.0	True	106.173316	44.811540	2.0	5.0
decision_tree	8.0	True	-16.366139	34.413242	2.0	5.0
linear_tree_logistic	6.0	True	122.876453	74.004758	2.0	10.0
DDQN	NaN	NaN	112.160324	50.817801	2.0	10.0
linear_tree_ridge	4.0	True	76.415637	102.956517	2.0	10.0
decision_tree	8.0	False	-24.005035	34.353172	2.0	10.0
linear_tree_logistic	6.0	True	89.437852	77.458409	2.0	15.0
DDQN	NaN	NaN	73.259875	52.894412	2.0	15.0
linear_tree_ridge	8.0	False	52.932146	101.607607	2.0	15.0
decision_tree	8.0	False	-25.006666	34.986570	2.0	15.0
DDQN	NaN	NaN	61.493178	51.995556	2.0	20.0
linear_tree_logistic	6.0	True	43.521199	74.454082	2.0	20.0
linear_tree_ridge	4.0	True	41.232509	101.044500	2.0	20.0
decision_tree	8.0	False	-73.580901	34.883665	2.0	20.0

No nossa análise, ao invés de apresentar os valores absolutos das recompensas médias, optamos por expressar o desempenho de cada modelo como uma fração ou múltiplo da recompensa média obtida pelo modelo especialista. Isso facilita a comparação entre as recompensas médias obtidas pelos diferentes tipos de modelo. Por exemplo, se um modelo obteve recompensas de 1.1x, quer dizer que conseguiu 10% mais recompensas que o DDQN.

CartPole

No ambiente CartPole, o modelo linear com regressão Ridge de profundidade 1, com reamostragem pesada, obteve a maior recompensa média com uma força do vento de 0,1, alcançando recompesas médias de aproximadamente 1.063x. Em seguida, o modelo linear com regressão logística de profundidade 2, também com reamostragem pesada, teve um desempenho próximo, com 1.005x. Isso demonstra que em condições de vento leve, as árvores lineares podem superar as redes neurais em termos de recompensa média.

À medida que a força do vento aumentou para 0,5, o modelo DDQN se demonstrou mais robusto, obtendo a recompensa mais alta entre os modelos testados. Os modelos lineares com regressão também apresentaram um desempenho razoável em condições de vento mais forte. A árvore linear com regressão logística de profundidade 1, sem reamostragem pesada, obteve uma recompensa média de 0.934x. A árvore linear com regressão Ridge de profundidade 6, sem reamostragem pesada, obteve uma recompensa de 0.787x. Esses resultados mostram que, em condições de vento mais forte previamente não observadas, os modelos lineares ainda conseguem manter uma performance relativamente boa em comparação ao modelo especialista. Teorizamos que como este ambiente contém apenas duas ações e quatro variáveis observáveis, árvores lineares de baixa profundidade já são capazes de entender sua dinâmica. Adicionalmente, o processo de reponderação do VIPER tende a não ajudar já que a simplicidade do ambiente não requer ajustes complexos para capturar a política ótima.

As árvores de decisão simples apresentaram um desempenho relativamente inferior em condições de vento leve. Com uma força do vento de 0,1, as árvores de decisão alcançaram recompensas médias de 0.768x para profundidade 6 e 0.721x para profundidade 4, posicionando-se atrás dos modelos lineares e do DDQN. À medida que a força do vento aumentou, o desempenho das árvores de decisão simples melhorou em termos relativos. Com uma força do vento de 0,2, as recompensas médias das árvores de decisão foram para 0.933x (profundidade 8) e 0.933x (profundidade 6), ainda atrás dos modelos lineares e do DDQN.

Em condições de vento mais extremas, as recompensas médias das árvores de decisão simples continuaram a cair. Com uma força do vento de 0,3, as árvores de decisão simples tiveram recompensas médias de 0.887x para profundidade 6. Com uma força do vento de 0,4, as recompensas médias foram de 0.843x para profundidade 8 e 0.833x para profundidade 6. Com uma força do vento de 0,5, as recompensas médias foram 0.787x para profundidade 6 e 0.721x para profundidade 8. Embora as árvores de decisão simples não sejam as mais eficazes em condições mais adversas do CartPole robusto, elas ainda mantêm um desempenho razoável, especialmente em cenários de vento mais intenso.

LunarLander

No ambiente LunarLander, a análise mostra que os modelos lineares com regressão logística e Ridge frequentemente superaram o modelo DDQN,

especialmente em condições de vento mais amenas. Com força do vento nula, a árvore linear com regressão logística de profundidade 6, com reamostragem pesada, alcançou a maior recompensa média de aproximadamente 1.150x. Em seguida, a árvore linear com regressão logística de profundidade 1, sem reamostragem pesada, obteve uma recompensa média de 1.166x, mostrando que em condições de alta turbulência mas sem vento, as árvores lineares conseguem ultrapassar o especialista.

À medida que a força do vento aumentou para 5, a árvore linear com regressão logística de profundidade 1, sem reamostragem pesada, manteve um desempenho superior, com uma recompensa média de aproximadamente 1.166x. Com uma força do vento de 10, a árvore linear com regressão logística de profundidade 6, com reamostragem pesada, obteve a maior recompensa média, de 1.095x.

Em forças do vento mais extremas, como 15 e 20, o desempenho do modelo DDQN caiu, e as árvores lineares ainda mantiveram recompensas médias superiores em alguns casos. Por exemplo, com uma força do vento de 15, a árvore linear com regressão logística de profundidade 6, com reamostragem pesada, alcançou uma recompensa média de 1.221x. Com uma força do vento de 20, a árvore linear com regressão logística de profundidade 6, com reamostragem pesada, alcançou uma recompensa média de 0.708x. Isto demontra que mesmo nas condições mais extremas dentre as propostas, as árvores lineares ainda podem ser competitivas.

Tanto no ambiente CartPole quanto no LunarLander, os modelos de árvores lineares mostraram-se capazes de superar o modelo DDQN em várias condições de vento, especialmente em forças de vento leves e moderadas. No entanto, o modelo DDQN ainda se manteve robusto e eficaz, especialmente em condições de vento extremo. Esses resultados sugerem que as árvores lineares podem ser uma alternativa viável e eficiente em termos de tempo de execução, mas a eficácia pode variar dependendo das condições específicas do ambiente e da força do vento.

Os boxplots das Figuras 4.3 e 4.4 ilustram a dispersão e a mediana das recompensas ao longo de 1000 episódios. As distribuições da Figura 4.3 foram geradas para um ambiente sem vento e as distribuições da Figura 4.4 foram geradas para um ambiente com vento e turbulência. O círculo branco em cada caixa representa a média das recompensas para o agente correspondente.

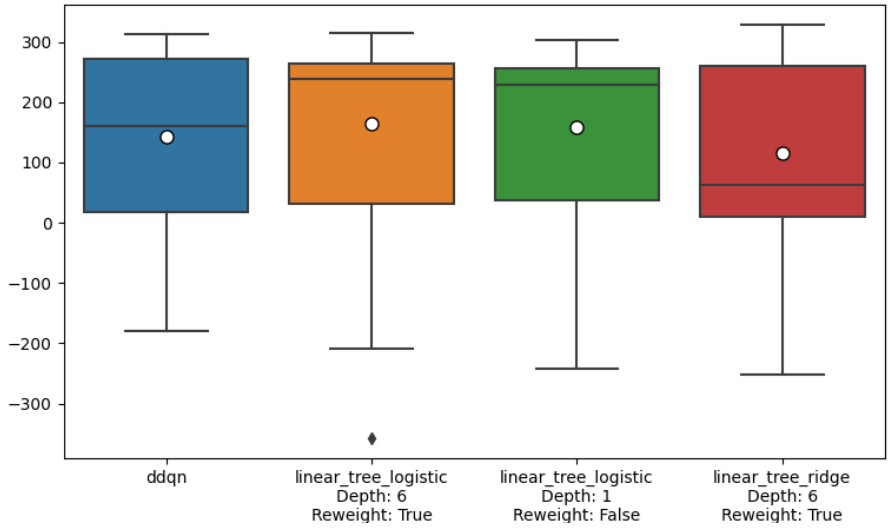


Figura 4.3: Distribuição das recompensas em 1000 episódios - ambiente sem vento
(enable_wind=False)

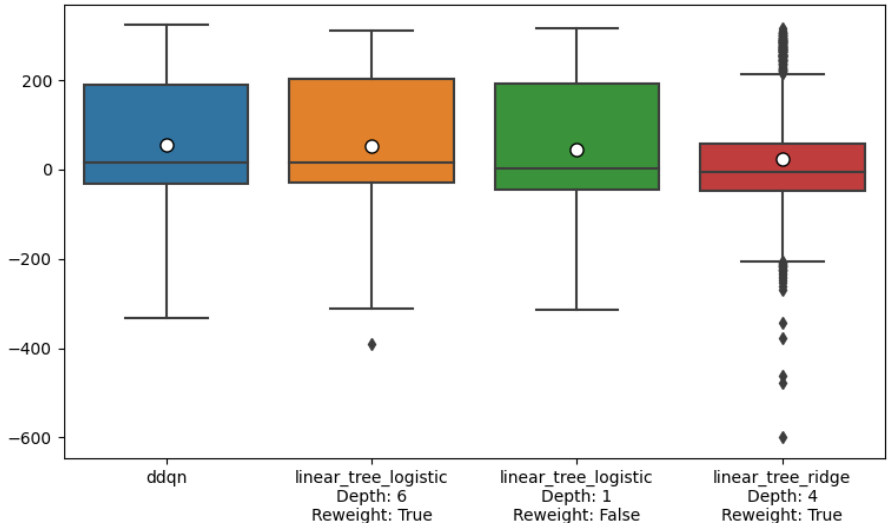


Figura 4.4: Distribuicão das recompensas em 1000 episódios - ambiente com vento
(enable_wind=True, wind_power=20, turbulence_power = 1.5)

Pode-se observar que no ambiente sem vento, a `logistic_dt` obtém mais recompensas que os outros modelos, e com menos dispersão. No ambiente com vento no valor experimental máximo e turbulência, sua média fica abaixo da DDQN. Uma interpretação plausível deste decaimento é que a rede neural aprendeu uma política mais robusta. Isto faz sentido, visto que é treinada com mais informação do que as árvores — a rede recebe também as recompensas

para cada estado e avalia possíveis interações com o ambiente a partir da sua função-Q.

5
Conclusão e trabalhos futuros

Neste estudo, aplicamos técnicas de aprendizado por reforço e aprendizado por imitação em diversos ambientes do Farama Gymnasium. Usando uma DDQN como modelo especialista, treinamos diferentes tipos de árvore de decisão como aprendizes, com duas variações de um algoritmo iterativo que busca treinar os aprendizes numa base de dados representativa da tarefa. Notavelmente, alguns agentes treinados via DAgger/VIPER se aproximaram, ou às vezes até superaram, o desempenho do especialista. Para explicar esses resultados, são necessárias mais análises e uma revisão da literatura. Este foi um resultado supreendente da análise – inicialmente pensamos que o desempenho de um aprendiz estaria limitado pelo do especialista. Uma futura direção para esta linha de pesquisa seria explicar em quais condições essa melhoria ocorre — uma hipótese é que as árvores de modelos lineares podem estar suavizando a fronteira de decisão do DDQN, evitando assim o *overfitting* e gerando políticas mais robustas para certas condições.

Em ambientes sem vento ou com vento leve, os modelos lineares com regressão logística e Ridge frequentemente superaram o DDQN. Este comportamento é indicativo de que as árvores de decisão, ao incorporar modelos lineares em suas folhas, conseguem capturar de maneira eficaz as dinâmicas do ambiente em condições menos adversas. No entanto, à medida que a força do vento aumentou, o DDQN demonstrou ser mais robusto, mantendo o melhor desempenho em condições mais adversas. Isso sugere que o DDQN aprendeu políticas que se generalizaram melhor em cenários com condições extremas, onde a complexidade e a variabilidade do ambiente impõem desafios maiores ao modelo. Por outro lado, as árvores de modelos lineares mostraram-se eficientes em termos de tempo de execução e desempenho, especialmente em condições mais controladas, evidenciando sua potencialidade em ambientes estáveis.

Outras técnicas, como aquelas baseadas no aprendizado de árvores a partir de interação direta com ambiente, sem necessidade de ser ocasionalmente corrigido por um modelo especialista (LIU et al., 2018), deveriam ser exploradas em trabalhos futuros para oferecer mais pontos comparativos. Além disso, a investigação do impacto de outras condições atmosféricas além do vento, como variações na gravidade e a introdução de turbulências mais complexas, poderá fornecer insights adicionais sobre a robustez das políticas aprendidas. Esse tipo de análise poderia revelar a resiliência dos agentes em um espectro mais amplo de condições ambientais e ajudar a desenvolver modelos mais

adaptativos e robustos.

Adicionalmente, futuros trabalhos poderiam considerar a integração de diferentes técnicas de regularização durante o treinamento das árvores de decisão para mitigar problemas de overfitting, bem como explorar a hibridização de modelos, combinando a capacidade de generalização das redes neurais com a interpretabilidade das árvores de decisão (FROSST; HINTON, 2017). A análise comparativa de outras abordagens tangentes ao aprendizado por imitação, como *Offline RL* que dependem únicamente de uma base de dados com demonstrações diversas o suficiente, também seria benéfica para aprofundar as vantagens e limitações das técnicas utilizadas.

Devemos enfatizar que nossos resultados são contextuais e baseados no ambiente e nas configurações que escolhemos. Não podemos assumir que as mesmas abordagens e estratégias funcionarão igualmente bem em outros contextos ou cenários além dos estipulados. Portanto, é crucial conduzir experimentos adicionais em uma variedade de ambientes para validar a generalidade das conclusões obtidas e identificar potenciais adaptações necessárias para diferentes tipos de problemas. A aplicação dessas técnicas em cenários do mundo real também seria um passo importante para avaliar a viabilidade prática e a eficácia dos modelos desenvolvidos.

6
Resultados adicionais

Tabela 6.1: Resultados (CartPole-v1)

env_name	model_type	max_depth	is_reweight	avg_reward	avg_time
CartPole-v1	DDQN	NaN	NaN	500.000000	42.808232
CartPole-v1	decision_tree	1.0	False	15.347000	27.383146
CartPole-v1	decision_tree	1.0	True	41.817000	26.831458
CartPole-v1	decision_tree	2.0	False	127.993000	26.432818
CartPole-v1	decision_tree	2.0	True	268.611000	26.185332
CartPole-v1	decision_tree	4.0	False	469.256000	26.632576
CartPole-v1	decision_tree	4.0	True	253.870000	26.392104
CartPole-v1	decision_tree	6.0	False	399.166000	26.728565
CartPole-v1	decision_tree	6.0	True	284.414000	26.650983
CartPole-v1	decision_tree	8.0	False	448.221000	26.254955
CartPole-v1	decision_tree	8.0	True	320.458000	26.683795
CartPole-v1	linear_tree_logistic	1.0	False	499.281000	13.525307
CartPole-v1	linear_tree_logistic	1.0	True	495.433000	13.762913
CartPole-v1	linear_tree_logistic	2.0	False	500.000000	24.434292
CartPole-v1	linear_tree_logistic	2.0	True	500.000000	19.299410
CartPole-v1	linear_tree_logistic	4.0	False	500.000000	33.270020
CartPole-v1	linear_tree_logistic	4.0	True	500.000000	37.760238
CartPole-v1	linear_tree_logistic	6.0	False	499.192000	55.883301
CartPole-v1	linear_tree_logistic	6.0	True	500.000000	48.460407
CartPole-v1	linear_tree_logistic	8.0	False	500.000000	50.363066
CartPole-v1	linear_tree_logistic	8.0	True	499.983000	47.221210
CartPole-v1	linear_tree_ridge	1.0	False	499.740000	33.944014
CartPole-v1	linear_tree_ridge	1.0	True	500.000000	33.566142
CartPole-v1	linear_tree_ridge	2.0	False	500.000000	38.832968
CartPole-v1	linear_tree_ridge	2.0	True	500.000000	44.526656
CartPole-v1	linear_tree_ridge	4.0	False	500.000000	52.996922
CartPole-v1	linear_tree_ridge	4.0	True	500.000000	53.490560
CartPole-v1	linear_tree_ridge	6.0	False	495.669000	73.127545
CartPole-v1	linear_tree_ridge	6.0	True	441.337000	73.827459
CartPole-v1	linear_tree_ridge	8.0	False	498.652000	61.807833
CartPole-v1	linear_tree_ridge	8.0	True	499.818000	67.430079

Tabela 6.2: Resultados (LunarLander-v2)

env_name	model_type	max_depth	is_reweight	avg_reward	avg_time
LunarLander-v2	DDQN	NaN	NaN	257.970973	48.717298
LunarLander-v2	decision_tree	1.0	False	-111.056514	31.488531
LunarLander-v2	decision_tree	1.0	True	-149.838758	31.619444
LunarLander-v2	decision_tree	2.0	False	-41.372616	31.393816
LunarLander-v2	decision_tree	2.0	True	-45.306850	30.885035
LunarLander-v2	decision_tree	4.0	False	-225.177639	30.797033
LunarLander-v2	decision_tree	4.0	True	-41.696215	29.714278
LunarLander-v2	decision_tree	6.0	False	-76.341881	30.548786
LunarLander-v2	decision_tree	6.0	True	-55.643926	30.623425
LunarLander-v2	decision_tree	8.0	False	29.449678	30.590557
LunarLander-v2	decision_tree	8.0	True	44.775291	30.754009
LunarLander-v2	linear_tree_logistic	1.0	False	190.345433	15.406075
LunarLander-v2	linear_tree_logistic	1.0	True	206.923609	15.353279
LunarLander-v2	linear_tree_logistic	2.0	False	197.964999	28.007708
LunarLander-v2	linear_tree_logistic	2.0	True	194.090302	27.745100
LunarLander-v2	linear_tree_logistic	4.0	False	184.964763	57.790583
LunarLander-v2	linear_tree_logistic	4.0	True	252.330563	50.923496
LunarLander-v2	linear_tree_logistic	6.0	False	205.262065	50.033985
LunarLander-v2	linear_tree_logistic	6.0	True	228.858184	58.217474
LunarLander-v2	linear_tree_logistic	8.0	False	195.257736	59.509749
LunarLander-v2	linear_tree_logistic	8.0	True	187.720430	60.387932
LunarLander-v2	linear_tree_ridge	1.0	False	203.068736	38.130497
LunarLander-v2	linear_tree_ridge	1.0	True	151.468399	38.243224
LunarLander-v2	linear_tree_ridge	2.0	False	158.540322	50.531935
LunarLander-v2	linear_tree_ridge	2.0	True	188.401532	51.196417
LunarLander-v2	linear_tree_ridge	4.0	False	189.541005	74.002258
LunarLander-v2	linear_tree_ridge	4.0	True	152.390040	81.502958
LunarLander-v2	linear_tree_ridge	6.0	False	143.449616	83.881242
LunarLander-v2	linear_tree_ridge	6.0	True	225.913827	82.674857
LunarLander-v2	linear_tree_ridge	8.0	False	139.446253	80.305498
LunarLander-v2	linear_tree_ridge	8.0	True	176.642679	73.067624

Tabela 6.3: Resultados (Taxi-v3)

env_name	model_type	max_depth	is_reweight	avg_reward	avg_time
Taxi-v3	DDQN	NaN	NaN	7.940000	48.295859
Taxi-v3	decision_tree	1.0	False	-200.000000	27.508261
Taxi-v3	decision_tree	1.0	True	-200.000000	27.892358
Taxi-v3	decision_tree	2.0	False	-200.000000	27.454744
Taxi-v3	decision_tree	2.0	True	-200.000000	27.883908
Taxi-v3	decision_tree	4.0	False	-285.986000	27.174042
Taxi-v3	decision_tree	4.0	True	-574.283000	27.292606
Taxi-v3	decision_tree	6.0	False	-200.000000	27.121848
Taxi-v3	decision_tree	6.0	True	-224.446000	27.556379
Taxi-v3	decision_tree	8.0	False	-101.157000	27.339854
Taxi-v3	decision_tree	8.0	True	-95.010000	27.526508
Taxi-v3	linear_tree_logistic	1.0	False	-346.988000	12.949798
Taxi-v3	linear_tree_logistic	1.0	True	-411.194000	12.895302
Taxi-v3	linear_tree_logistic	2.0	False	7.736000	23.252857
Taxi-v3	linear_tree_logistic	2.0	True	7.912000	23.257703
Taxi-v3	linear_tree_logistic	4.0	False	7.904000	41.454541
Taxi-v3	linear_tree_logistic	4.0	True	8.002000	34.873965
Taxi-v3	linear_tree_logistic	6.0	False	-266.346000	34.270350
Taxi-v3	linear_tree_logistic	6.0	True	7.743000	34.970056
Taxi-v3	linear_tree_logistic	8.0	False	7.781000	34.646759
Taxi-v3	linear_tree_logistic	8.0	True	7.945000	41.724709
Taxi-v3	linear_tree_ridge	1.0	False	-652.808000	33.548228
Taxi-v3	linear_tree_ridge	1.0	True	-200.000000	33.421810
Taxi-v3	linear_tree_ridge	2.0	False	-29.122000	45.043977
Taxi-v3	linear_tree_ridge	2.0	True	-418.079000	43.994174
Taxi-v3	linear_tree_ridge	4.0	False	5.558000	56.763811
Taxi-v3	linear_tree_ridge	4.0	True	0.022000	70.470901
Taxi-v3	linear_tree_ridge	6.0	False	-27.136000	56.865935
Taxi-v3	linear_tree_ridge	6.0	True	-35.883000	56.858861
Taxi-v3	linear_tree_ridge	8.0	False	-27.436000	56.923534
Taxi-v3	linear_tree_ridge	8.0	True	-19.088000	65.472609

Tabela 6.4: Resultados (Windy CartPole - Top 3 de cada modelo por configuração de vento)

model_type	max_depth	is_reweight	avg_reward	avg_time	turbulence_power	wind_power
linear_tree_ridge	1.0	True	315.585	56.564687	0.1	0.1
linear_tree_logistic	2.0	True	298.251	45.959514	0.1	0.1
DDQN	NaN	NaN	296.794	32.202289	0.1	0.1
linear_tree_logistic	4.0	True	296.747	65.364538	0.1	0.1
linear_tree_logistic	1.0	False	274.162	36.973830	0.1	0.1
linear_tree_ridge	4.0	True	273.270	86.418375	0.1	0.1
linear_tree_ridge	2.0	False	271.412	64.196592	0.1	0.1
decision_tree	6.0	False	227.880	24.657906	0.1	0.1
decision_tree	4.0	False	213.967	24.831685	0.1	0.1
decision_tree	8.0	False	209.170	24.790407	0.1	0.1
linear_tree_logistic	1.0	False	173.521	39.763403	0.1	0.2
DDQN	NaN	NaN	163.474	32.738032	0.1	0.2
linear_tree_ridge	1.0	True	161.135	61.299682	0.1	0.2
linear_tree_ridge	2.0	False	160.462	66.818233	0.1	0.2
linear_tree_ridge	6.0	False	160.299	104.851842	0.1	0.2
linear_tree_logistic	4.0	True	155.315	67.899240	0.1	0.2
decision_tree	8.0	False	152.502	25.124091	0.1	0.2
decision_tree	6.0	False	152.144	25.201023	0.1	0.2
linear_tree_logistic	1.0	True	149.476	39.037327	0.1	0.2
decision_tree	6.0	True	146.264	24.930870	0.1	0.2
linear_tree_logistic	1.0	False	152.362	38.198518	0.1	0.3
DDQN	NaN	NaN	145.281	33.225942	0.1	0.3
decision_tree	6.0	True	128.894	25.544641	0.1	0.3
linear_tree_ridge	6.0	False	126.940	104.131813	0.1	0.3
decision_tree	6.0	False	126.829	25.730537	0.1	0.3
decision_tree	8.0	False	126.594	25.423325	0.1	0.3
linear_tree_ridge	4.0	False	123.701	82.218814	0.1	0.3
linear_tree_logistic	4.0	True	120.923	64.718722	0.1	0.3
linear_tree_logistic	4.0	False	115.096	59.926423	0.1	0.3
linear_tree_ridge	8.0	False	106.663	91.562045	0.1	0.3
linear_tree_logistic	1.0	False	124.863	38.776649	0.1	0.4
DDQN	NaN	NaN	117.432	33.431244	0.1	0.4
linear_tree_ridge	4.0	False	98.959	81.857060	0.1	0.4
decision_tree	8.0	False	97.861	26.085891	0.1	0.4
linear_tree_ridge	6.0	False	96.377	105.171393	0.1	0.4
linear_tree_logistic	4.0	True	94.009	66.588668	0.1	0.4
decision_tree	6.0	True	92.379	26.064274	0.1	0.4
decision_tree	6.0	False	92.090	26.083787	0.1	0.4
linear_tree_logistic	4.0	False	89.164	60.656100	0.1	0.4
linear_tree_ridge	2.0	False	89.067	65.524108	0.1	0.4
DDQN	NaN	NaN	95.036	33.473905	0.1	0.5
linear_tree_logistic	1.0	False	88.788	38.387450	0.1	0.5
decision_tree	6.0	True	79.516	25.960644	0.1	0.5
decision_tree	8.0	False	78.594	25.714765	0.1	0.5
decision_tree	6.0	False	77.226	25.792051	0.1	0.5
linear_tree_logistic	4.0	False	75.049	61.183205	0.1	0.5
linear_tree_ridge	6.0	False	74.860	103.359457	0.1	0.5
linear_tree_logistic	4.0	True	72.789	66.027605	0.1	0.5
linear_tree_ridge	2.0	False	70.753	66.207052	0.1	0.5
linear_tree_ridge	4.0	False	70.712	81.667773	0.1	0.5

Tabela 6.5: Resultados (LunarLander, vento e turbulência - Top 3 de cada modelo por configuração de vento)

model_type	max_depth	is_reweight	avg_reward	avg_time	turbulence_power	wind_power
linear_tree_logistic	6.0	True	159.699392	75.211113	2.0	0.0
linear_tree_logistic	1.0	False	153.773067	19.136055	2.0	0.0
DDQN	NaN	NaN	138.887504	51.795167	2.0	0.0
linear_tree_logistic	8.0	False	132.609315	76.855215	2.0	0.0
linear_tree_ridge	6.0	True	115.281745	103.952109	2.0	0.0
linear_tree_ridge	1.0	True	104.153795	45.266234	2.0	0.0
linear_tree_ridge	8.0	True	93.854067	89.203569	2.0	0.0
decision_tree	8.0	True	-8.431982	34.796988	2.0	0.0
decision_tree	8.0	False	-16.876786	34.909518	2.0	0.0
decision_tree	6.0	True	-30.831514	34.479974	2.0	0.0
linear_tree_logistic	1.0	False	139.191163	18.775421	2.0	5.0
linear_tree_logistic	6.0	True	134.327613	74.532913	2.0	5.0
linear_tree_logistic	8.0	True	133.112860	78.796169	2.0	5.0
DDQN	NaN	NaN	119.409141	51.027988	2.0	5.0
linear_tree_ridge	1.0	True	106.173316	44.811540	2.0	5.0
linear_tree_ridge	4.0	True	100.926087	100.018626	2.0	5.0
linear_tree_ridge	6.0	True	90.591474	102.470488	2.0	5.0
decision_tree	8.0	True	-16.366139	34.413242	2.0	5.0
decision_tree	8.0	False	-23.663868	34.955338	2.0	5.0
decision_tree	6.0	True	-34.919344	34.424009	2.0	5.0
linear_tree_logistic	6.0	True	122.876453	74.004758	2.0	10.0
linear_tree_logistic	1.0	False	119.050033	18.824398	2.0	10.0
DDQN	NaN	NaN	112.160324	50.817801	2.0	10.0
linear_tree_logistic	8.0	True	109.221938	76.993655	2.0	10.0
linear_tree_ridge	4.0	True	76.415637	102.956517	2.0	10.0
linear_tree_ridge	6.0	True	69.894632	106.737943	2.0	10.0
linear_tree_ridge	1.0	True	68.302651	44.584205	2.0	10.0
decision_tree	8.0	False	-24.005035	34.353172	2.0	10.0
decision_tree	6.0	True	-32.672162	33.940297	2.0	10.0
decision_tree	8.0	True	-40.751834	34.089241	2.0	10.0
linear_tree_logistic	6.0	True	89.437852	77.458409	2.0	15.0
linear_tree_logistic	8.0	True	82.476852	80.510824	2.0	15.0
DDQN	NaN	NaN	73.259875	52.894412	2.0	15.0
linear_tree_ridge	8.0	False	52.932146	101.607607	2.0	15.0
linear_tree_ridge	2.0	True	51.992155	62.898286	2.0	15.0
linear_tree_logistic	4.0	True	50.426696	67.847860	2.0	15.0
linear_tree_ridge	4.0	True	48.741345	101.321028	2.0	15.0
decision_tree	8.0	False	-25.006666	34.986570	2.0	15.0
decision_tree	8.0	True	-36.964168	35.177041	2.0	15.0
decision_tree	6.0	True	-65.069217	34.587414	2.0	15.0
DDQN	NaN	NaN	61.493178	51.995556	2.0	20.0
linear_tree_logistic	6.0	True	43.521199	74.454082	2.0	20.0
linear_tree_logistic	1.0	False	42.328048	18.772220	2.0	20.0
linear_tree_logistic	4.0	True	42.263520	65.245044	2.0	20.0
linear_tree_ridge	4.0	True	41.232509	101.044500	2.0	20.0
linear_tree_ridge	2.0	False	27.806370	60.982702	2.0	20.0
linear_tree_ridge	8.0	False	26.838750	98.837465	2.0	20.0
decision_tree	8.0	False	-73.580901	34.883665	2.0	20.0
decision_tree	6.0	True	-73.656783	34.561875	2.0	20.0
decision_tree	8.0	True	-79.200419	35.036346	2.0	20.0

7
Referências bibliográficas

BASTANI, O.; PU, Y.; SOLAR-LEZAMA, A. Verifiable reinforcement learning via policy extraction. In: **Proceedings of the 35th International Conference on Machine Learning (ICML)**. [S.l.: s.n.], 2018. Citado 3 vezes nas páginas 17, 18 e 19.

BREIMAN, L. et al. **Classification and Regression Trees**. [S.l.]: Chapman and Hall/CRC, 1984. Citado 2 vezes nas páginas 6 e 29.

BUCILA, C.; CARUANA, R.; NICULESCU-MIZIL, A. Model compression. **Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining**, p. 535–541, 2006. Citado na página 13.

CERLIANI, M. **linear-tree: A Python library for building linear decision trees**. 2024. Accessed: 2023-12-11. Disponível em: <https://github.com/cerlymarco/linear-tree>. Citado na página 29.

DOSHI-VELEZ, F.; KIM, B. Towards a rigorous science of interpretable machine learning. **arXiv preprint arXiv:1702.08608**, 2017. Citado na página 6.

EDRAWMAX. **What is a Decision Tree?** 2024. Accessed: 2023-12-11. Disponível em: <https://www.edrawmax.com/decision-tree/>. Citado na página 6.

FROSST, N.; HINTON, G. Distilling a neural network into a soft decision tree. p. 1–2, 11 2017. Citado 2 vezes nas páginas 6 e 43.

HASSELT, H. van; GUEZ, A.; SILVER, D. **Deep Reinforcement Learning with Double Q-learning**. 2015. Google DeepMind. Citado 2 vezes nas páginas 13 e 19.

KATZ, G. et al. Reluplex: An efficient SMT solver for verifying deep neural networks. In: MAJUMDAR, R.; KUNCAK, V. (Ed.). **Proceedings of the 29th International Conference on Computer Aided Verification (CAV '17)**. Heidelberg, Germany: Springer, 2017. (Lecture Notes in Computer Science, v. 10426), p. 97–117. Disponível em: <http://theory.stanford.edu/~barrett/pubs/KBD+17.pdf>. Citado na página 6.

LAMPTON, A.; VALASEK, J. Multiresolution state-space discretization method for q-learning with function approximation and policy iteration. In: IEEE. **2009 American Control Conference**. 2009. p. 5106–5111. Disponível em: <https://ieeexplore.ieee.org/document/5346129>. Citado na página 5.

LEVINE, S. et al. Offline reinforcement learning: Tutorial, review, and perspectives on open problems. **ArXiv**, abs/2005.01643, 2020. Disponível em: <https://api.semanticscholar.org/CorpusID:218486979>. Citado na página 19.

LIU, G. et al. Toward interpretable deep reinforcement learning with linear model u-trees. In: **European Conference on Machine Learning and Principles and**

Practice of Knowledge Discovery in Databases (ECML PKDD). [S.l.: s.n.], 2018. Citado 2 vezes nas páginas 19 e 42.

MNIH, V. et al. Human-level control through deep reinforcement learning. **Nature**, Nature Publishing Group, v. 518, n. 7540, p. 529–533, 2015. Citado na página 11.

MNIH, V. et al. **Playing Atari with Deep Reinforcement Learning**. 2013. DeepMind Technologies. Citado 3 vezes nas páginas 5, 12 e 19.

POMERLEAU, D. Efficient training of artificial neural networks for autonomous navigation. **Neural Computation**, v. 3, p. 88–97, 1991. Citado 2 vezes nas páginas 12 e 14.

ROSS, S.; BAGNELL, D. Efficient reductions for imitation learning. In: PMLR. **Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics**. [S.l.], 2010. p. 661–668. Citado 2 vezes nas páginas 14 e 15.

ROSS, S.; GORDON, G. J.; BAGNELL, J. A. A reduction of imitation learning and structured prediction to no-regret online learning. In: **Proceedings of the 14th International Conference on Artificial Intelligence and Statistics (AISTATS)**. [S.l.: s.n.], 2011. Citado 3 vezes nas páginas 14, 15 e 16.

SUTTON, R.; BARTO, A. **Reinforcement Learning: An Introduction**. [S.l.]: The MIT Press, 2018. Citado 4 vezes nas páginas 5, 8, 9 e 10.

TOWERS, M. et al. **Gymnasium**. 2023. Disponível em: <https://github.com/Farama-Foundation/Gymnasium>. Citado 4 vezes nas páginas 21, 22, 23 e 25.

WATKINS, C. **Learning from delayed rewards**. Tese (Doutorado) — King's College, 1989. Citado 2 vezes nas páginas 5 e 10.

WATKINS, C.; DAYAN, P. **Q-Learning**. [S.l.], 1992. Citado na página 11.