Estacionamento Autônomo por Otimização Evolutiva com Redes Neurais em Ambiente Simulado

Universidade Federal de São Paulo – UNIFESP
Instituto de Ciência e Tecnologia – Campus São José dos Campos

Micael de Oliveira Pimpim
RA: 120875

SelfParking: Estacionamento Autônomo por Otimização Evolutiva com Redes Neurais em Ambiente Simulado

Jacareí – SP
2025

Folha de Rosto

Micael de Oliveira Pimpim – RA: 120875

SelfParking: Estacionamento Autônomo por Otimização Evolutiva com Redes Neurais em Ambiente Simulado

Monografia apresentada à Universidade Federal de São Paulo – UNIFESP, como parte dos requisitos para obtenção do título de **Engenheiro de Computação**.

Orientador: Prof. Dr. [Nome do Orientador]

São José dos Campos – SP
2025

Resumo

O presente trabalho desenvolve o **SelfParking**, um sistema de estacionamento autônomo utilizando técnicas de otimização evolutiva integradas a redes neurais artificiais em um ambiente de simulação. O objetivo principal é capacitar um agente veicular (carro) a realizar manobras de estacionamento de forma autônoma, aprendendo a controlar a direção e velocidade sem intervenção humana direta. Para isso, empregou-se uma **rede neural do tipo Perceptron Multicamadas (MLP)** como controlador de baixíssimo nível, responsável por tomar decisões de controle contínuo a partir de sensores do veículo. Os pesos dessa rede neural são treinados por um **Algoritmo Genético (AG)**, que evolui sucessivas gerações de controladores candidatos com base em um critério de recompensa definido pela qualidade da manobra de estacionamento. O trabalho aborda conceitos fundamentais de controle automatizado e autônomo, redes neurais artificiais e algoritmos genéticos, descrevendo a arquitetura do sistema desenvolvido na engine Unity3D, o design do

ambiente simulado e as métricas de desempenho utilizadas. Os resultados parciais demonstram que o agente evolutivo consegue **aprender estratégias de estacionamento** eficazes, obtendo 100% de sucesso em estacionar dentro da vaga e sem colisões em cenários simulados variados. A **discussão crítica** destaca as vantagens e limitações da abordagem neuro-evolutiva, comparando-a com métodos clássicos de controle e aprendizado por reforço, e aponta melhorias e passos futuros para aumentar a robustez e aproximar o sistema de aplicações reais.

Palavras-chave: Controle Autônomo; Redes Neurais Artificiais; Algoritmos Genéticos; Aprendizado por Reforço; Estacionamento Automatizado.

Abstract

This work presents **SelfParking**, an autonomous parking system using evolutionary optimization techniques integrated with artificial neural networks in a simulated environment. The main goal is to enable a vehicle agent (car) to perform parking maneuvers autonomously, learning to control steering and speed without direct human intervention. For that, a **Multi-Layer Perceptron (MLP) neural network** was employed as a low-level controller, responsible for making continuous control decisions from the vehicle's sensors. The weights of this neural network are trained by a **Genetic Algorithm (GA)**, which evolves successive generations of candidate controllers based on a reward criterion defined by the parking maneuver quality. This work covers fundamental concepts of automated and autonomous control, artificial neural networks, and genetic algorithms, describing the architecture of the system developed in the Unity3D engine, the design of the simulated environment, and the performance metrics used. The partial results show that the evolutionary agent is able to **learn effective parking strategies**, achieving a 100% success rate in parking within the spot without collisions under varied simulated scenarios. The **critical discussion** highlights the advantages and limitations of the neuro-evolutionary approach, comparing it with classical control and reinforcement learning methods, and points out improvements and future steps to increase robustness and bring the system closer to real-world applications.

Keywords: Autonomous Control; Artificial Neural Networks; Genetic Algorithms; Reinforcement Learning; Automated Parking.

Sumário

---

1. Introdução

O desenvolvimento de veículos autônomos e sistemas de assistência avançada ao motorista tem alcançado grande interesse na última década. **Estacionar um veículo** com segurança e precisão é uma tarefa rotineira, porém desafiadora, especialmente em vagas apertadas ou com pouca visibilidade. *Sistemas de estacionamento automático* já estão presentes em veículos modernos, usando sensores ultrassônicos e câmeras para auxiliar o motorista ou até mesmo controlar o volante automaticamente. Contudo, tais sistemas geralmente dependem de abordagens de controle clássicas e programação determinística para detectar vagas e realizar manobras, podendo apresentar limitações frente a cenários não previstos ou mudanças no ambiente.

Diante desse contexto, este trabalho propõe uma abordagem baseada em **aprendizado de máquina** e **otimização evolutiva** para o problema de estacionar um

carro de forma autônoma. Em vez de projetar manualmente cada etapa da manobra, o sistema intitulado **SelfParking** busca *aprender* a estacionar por si só, através da interação em um ambiente simulado. Essa aprendizagem é conduzida por técnicas inspiradas na evolução natural, em que candidatos a controladores de estacionamento competem entre si e melhoram gradativamente por meio de seleção, cruzamento e mutação. Ao mesmo tempo, utilizam-se **redes neurais artificiais** para representar o controlador autônomo do veículo, dada sua capacidade de aproximar funções complexas e tomar decisões não lineares em tempo real.

A motivação principal para essa pesquisa reside em avaliar se a combinação de **Algoritmos Genéticos (AG)** com **Redes Neurais (RN)** pode gerar um controlador capaz de estacionar um veículo de forma autônoma, de maneira eficaz e adaptável. Essa combinação, frequentemente denominada *neuroevolução*, aproveita o poder de busca global e estocástica dos algoritmos genéticos com a capacidade de modelagem e generalização das redes neurais[1]. Espera-se que o sistema resultante apresente maior flexibilidade frente a variações de cenário, sem a necessidade de modelagem matemática explícita da cinemática do carro ou do desenho de controladores de forma analítica.

Os **objetivos específicos** deste trabalho incluem:

- **Implementar um ambiente simulado** realista de estacionamento, utilizando a engine Unity3D, no qual o agente (carro) possa interagir e receber medições de sensores e recompensas de desempenho.
- **Desenvolver uma arquitetura de agente autônomo** composta por sensores virtuais, uma rede neural de múltiplas camadas atuando como sistema de controle do veículo, e um algoritmo genético responsável por ajustar os pesos dessa rede.
- **Definir funções de recompensa e métricas de desempenho** adequadas que incentivem comportamentos desejáveis (estacionar centralizado na vaga, evitar colisões, minimizar tempo e movimentos) e penalizem comportamentos indesejáveis.
- **Avaliar a aprendizagem evolutiva** do agente, conduzindo experimentos de treinamento por múltiplas gerações, registrando a evolução do desempenho (fitness) e verificando se o agente de fato adquire a habilidade de estacionar confiavelmente.
- **Testar a generalização** do controlador treinado em condições distintas das de treinamento – por exemplo, iniciando de posições aleatórias, ou mudando a disposição das vagas e presença de veículos obstáculo – analisando o quão bem o agente adaptado consegue lidar com variações.
- **Comparar e discutir** os resultados obtidos com abordagens alternativas, destacando vantagens e desvantagens. Em especial, compara-se o método neuro-evolutivo com estratégias clássicas de controle (como controladores

PID ou baseados em trajetória pré-calculada) e também com métodos de aprendizado por reforço tradicional (onde a otimização é feita por gradiente).

A **estrutura desta monografia** está organizada da seguinte forma: no **Capítulo 2** apresenta-se o **Referencial Teórico**, cobrindo os conceitos de controle automatizado e autônomo, fundamentos de redes neurais artificiais (perceptron multicamadas, funções de ativação, aprendizado supervisionado e não supervisionado) e fundamentos de algoritmos genéticos (motivação biológica, operadores de seleção, cruzamento, mutação e elitismo). Ainda no Capítulo 2 discute-se a integração entre algoritmos genéticos e redes neurais para fins de otimização de controle, base conceitual do sistema proposto.

No **Capítulo 3 – Metodologia**, detalha-se o ambiente de simulação desenvolvido e as ferramentas utilizadas (Unity3D e eventualmente ML-Agents), e descreve-se minuciosamente a **arquitetura do sistema SelfParking**. Isso inclui a modelagem do agente veículo e seus sensores virtuais, a configuração da rede neural de controle e a implementação do algoritmo genético em código C#. São explicadas também as **funções de recompensa e métricas** empregadas para avaliar o desempenho de cada controlador candidato, bem como o procedimento de treinamento evolutivo (critérios de parada, tamanho da população, taxas de mutação, etc.). Ao final da metodologia, descrevem-se os métodos de **validação** – isto é, como foram conduzidos os testes com a melhor rede neural obtida, em diferentes cenários e sob condições de ruído/aleatoriedade para medir a robustez.

No **Capítulo 4 – Resultados**, apresentam-se os dados coletados e observações empíricas. É exibida a **evolução do desempenho** do agente ao longo das gerações, por meio de gráficos e tabelas, bem como exemplos qualitativos do comportamento do carro antes e depois do treinamento (por exemplo, trajetórias típicas sem controle vs. com o controlador evoluído). Também são fornecidos indicadores quantitativos, como a taxa de sucesso em estacionar na vaga, número médio de colisões evitadas e tempo médio para estacionar, comparando diferentes experimentos ou configurações quando pertinente.

O **Capítulo 5 – Discussão** realiza uma análise crítica dos resultados, discutindo as razões pelas quais o algoritmo alcançou (ou não) sucesso, as principais dificuldades encontradas durante o desenvolvimento (por exemplo, fenômenos de *stall* na evolução, sensibilidade a parâmetros, etc.) e situando os achados em relação à literatura existente. Compara-se a eficiência do método com possíveis abordagens alternativas: discute-se, por exemplo, em que medida um controlador clássico ajustado manualmente poderia ter desempenho similar, ou como o uso de um algoritmo de aprendizado por reforço baseado em gradiente (como Deep Q-Learning, PPO) se compararia em termos de esforço de parametrização e resultado final. São enumeradas as **limitações** atuais do sistema SelfParking, tais como dependência de simulação (gap para mundo real), escalabilidade para cenários mais complexos (estacionamento paralelo, múltiplos agentes), e gargalos computacionais (tempo de

treinamento possivelmente elevado). Finalmente, apontam-se **possíveis melhorias e trabalhos futuros**, incluindo ajustes na arquitetura da rede neural, uso de técnicas avançadas de neuroevolução (como NEAT), inclusão de sensores adicionais (câmera para visão computacional), e validação em hardware real ou simulação mais fidedigna.

Por fim, o **Capítulo 6 – Conclusão** sintetiza os principais resultados alcançados e responde aos objetivos propostos, avaliando o sucesso do SelfParking em aprender a estacionar autonomamente. Ressalta-se as contribuições do trabalho e o aprendizado proporcionado, bem como recomendações para desenvolvimentos subsequentes. Complementam a monografia as **Referências Bibliográficas**, formatadas conforme as normas, e os **Anexos**, que incluem trechos relevantes de código fonte do projeto implementado.

2. Referencial Teórico

Neste capítulo são revisados os conceitos fundamentais que embasam o desenvolvimento do sistema *SelfParking*. Inicia-se discutindo os princípios de **controle automatizado** e a distinção para o **controle autônomo**, contextualizando como técnicas de aprendizado diferem dos métodos clássicos. Em seguida, são apresentados os fundamentos de **Redes Neurais Artificiais**, com ênfase no perceptron multicamadas, nas funções de ativação e nos paradigmas de aprendizado supervisionado e não supervisionado, dado que esses conceitos são centrais para o controlador proposto. Após isso, discorre-se sobre **Algoritmos Genéticos**, incluindo sua inspiração na evolução natural e os principais operadores (seleção, cruzamento, mutação, elitismo). Por fim, aborda-se a **integração de algoritmos genéticos com redes neurais** para otimização de controle, apresentando trabalhos e argumentos da literatura que motivam a abordagem neuro-evolutiva adotada.

2.1 Controle Automatizado e Controle Autônomo

A área de **Sistemas de Controle** trata de técnicas para que um sistema dinâmico (a chamada *planta*) produza um comportamento desejado em sua saída a partir de certas entradas de controle. Nos **sistemas de controle automatizados** tradicionais, um engenheiro elabora um controlador (por exemplo, um controlador PID, ou de lógica fixa) com base em um modelo matemático do sistema, para regular variáveis como posição, velocidade, temperatura, etc., de modo que acompanhem referências desejadas. Esse controlador é desenhado e **pré-definido** conforme a tarefa específica, e sua estrutura não muda durante a operação – ou seja, não há aprendizagem envolvida. Métodos clássicos de controle assumem um modelo relativamente preciso do sistema e utilizam técnicas analíticas para garantir estabilidade e desempenho (tempo de resposta, erro em regime, etc.) dentro desse modelo[2]. Por exemplo, para controlar a direção de um veículo, um controlador PID

poderia ser ajustado se conhecermos as equações que relacionam ângulo do volante e trajetória do carro.

No **controle autônomo baseado em aprendizado**, por outro lado, não se determina *a priori* todas as regras de ação do controlador. Em vez disso, o sistema adquire a política de controle por experiência, seja por dados de demonstração ou por tentativa e erro. No contexto de **Aprendizado por Reforço (Reinforcement Learning, RL)**, por exemplo, um agente aprende uma política ótima interagindo com o ambiente e recebendo recompensas por ações bem-sucedidas. Nesse caso, a "estrutura de controle" é a própria política aprendida, tipicamente representada por uma função de valor ou por uma rede neural, e **não é projetada diretamente pelo engenheiro**, mas sim ajustada pelo algoritmo de aprendizado. Isso significa que, diferentemente do controle clássico onde definimos equações de controle fixas, em RL ou métodos evolutivos partimos de uma política inicial (aleatória ou genérica) e esta vai se **auto-ajustando** com base no feedback do desempenho.

Uma comparação clara é dada por Figueiredo & Rejaili (2018) quando discutem controle clássico vs. aprendizado por reforço: *"Em métodos de Controle Clássico uma estrutura de controle é predefinida para realizar uma tarefa de controle bem estabelecida em um sistema físico normalmente modelado e utilizado para o design do controlador... Do ponto de vista da APR (aprendizado por reforço), porém, o sistema de controle pode ser entendido como a política aprendida pelo agente durante a aprendizagem, e nesse caso não se realiza diretamente o design das ações de controle"*[3][4]. Em outras palavras, **no controle clássico** projetamos primeiro e depois analisamos o desempenho; **no controle autônomo por aprendizado** (seja RL ou algoritmos genéticos), lançamos o agente para aprender e então precisamos analisar a política resultante e sua viabilidade.

Vale notar que o termo **"controle autônomo"** implica que o sistema controlador possui um certo grau de *independência adaptativa*, conseguindo tomar decisões sem intervenção humana mesmo diante de condições inesperadas. Por exemplo, em uma planta química, um sistema de controle autônomo baseado em IA (*Inteligência Artificial*) conseguiu operar colunas de destilação por 35 dias consecutivos, reagindo a mudanças repentinas de ambiente onde antes apenas operadores humanos conseguiam atuar[5][6]. Esse caso prático relatado por Yokogawa & JSR (2022) demonstra que técnicas de aprendizado (no caso, aprendizado por reforço profundo) permitiram atingir cenários além da capacidade de métodos convencionais (PID/APC). No contexto deste trabalho, controle autônomo refere-se à capacidade do veículo estacionar sozinho combinando sensoriamento e ação sem um algoritmo fixo codificado manualmente, mas sim com um comportamento **aprendido** e adaptativo.

Para conectar com o problema de estacionamento: Um **método clássico** poderia envolver segmentar a manobra em etapas pré-definidas (andar para frente até certo ponto, esterçar o volante em um determinado ângulo, etc.), calibradas para um tipo de vaga específico. Isso funciona bem em condições ideais, mas pode falhar se a

posição inicial do carro variar ou se houver pequenos desvios. Já um **método de aprendizado** (autônomo) poderia permitir que o carro *descubra* a sequência de movimentos apropriada, potencialmente lidando melhor com variações, pois ele otimiza suas ações diretamente pelo resultado alcançado (entrar corretamente na vaga). A contrapartida é que métodos de aprendizado exigem mais **dados ou simulações** para treinar e não garantem formalmente estabilidade ou ótimo global – o desempenho depende da qualidade do aprendizado e da definição adequada da função de recompensa ou critério de sucesso.

Em suma, o **controle automatizado** tradicional se baseia em controle pré-programado e ajustado por especialistas, enquanto o **controle autônomo por aprendizado** busca comportamentos de controle a partir de dados ou interações, tendo o potencial de superar limitações de modelos imprecisos e reagir a situações complexas[7]. Este trabalho se insere na segunda abordagem: usamos técnicas de aprendizado (algoritmo genético + rede neural) para que o veículo aprenda a estacionar, comparando depois essa solução com aquilo que seria obtido via design clássico.

2.2 Redes Neurais Artificiais

Redes Neurais Artificiais (RNA) são modelos computacionais inspirados livremente no cérebro humano e no sistema nervoso biológico, compostos por unidades de processamento simples interconectadas, chamadas de neurônios artificiais[8]. Cada neurônio artificial realiza uma operação simples: recebe sinais de entrada (que podem ser dados ou saídas de outros neurônios), aplica uma função matemática – normalmente não-linear, chamada **função de ativação** – e produz um sinal de saída. As conexões entre neurônios possuem pesos associados, que modulam a influência de uma saída sobre a entrada de outro neurônio[9]. A arquitetura típica de uma rede neural organiza neurônios em **camadas**: uma camada de entrada (que recebe diretamente os dados ou estímulos do ambiente), uma ou mais **camadas ocultas** (que fazem processamento interno) e uma camada de saída (que fornece a resposta final do sistema)[10]. A figura abaixo ilustra esse conceito estruturado de rede neural multicamadas.

(Figura 2 – ilustrar um diagrama de uma rede neural feedforward simples, com nós dispostos em camadas: entrada, uma ou duas camadas ocultas e camada de saída, indicando as funções de ativação em cada neurônio)

Uma instância muito utilizada de RNA é o **Perceptron Multicamadas**, frequentemente abreviado como **MLP (Multi-Layer Perceptron)**. Trata-se de uma rede **feedforward** densa, onde cada neurônio de uma camada está conectado a todos os neurônios da camada seguinte[11][12]. Em um MLP típico, os sinais propagam-se dos neurônios de entrada para os de saída, passando pelas camadas intermediárias, *sem ciclos* (por isso “feedforward”). Os neurônios de cada camada calculam somas ponderadas de suas entradas e aplicam uma função de ativação. **Funções de ativação** comuns incluem a sigmoide logística, a tangente hiperbólica

(*tanh*) e a ReLU (*Rectified Linear Unit*), entre outras. A principal função da ativação não-linear é introduzir *não-linearidade* na rede, pois somente assim a RNA consegue aprender padrões complexos e não simplesmente comportar-se como uma combinação linear trivial[13][14]. Em resumo, sem funções de ativação não lineares, mesmo várias camadas seriam matematicamente equivalentes a um modelo linear único[15].

No contexto deste trabalho, a rede neural será usada para mapear as leituras dos sensores do carro (distâncias até obstáculos, ângulos, etc.) em comandos de controle (ângulo de direção e aceleração/freio). Isso configura um mapeamento **não-linear** que pode ser bastante complicado de obter por dedução analítica, mas a rede neural pode aproximá-lo **aprendendo a partir de exemplos** ou de interação simulada.

2.2.1 Perceptron Multicamadas e Aprendizado Supervisionado

O perceptron multicamadas ganhou destaque por ser capaz de resolver problemas não linearmente separáveis clássicos (como a porta XOR), superando limitações do neurônio perceptron simples. A força de um MLP reside em suas camadas ocultas: elas constroem *representações internas* dos dados de entrada, frequentemente de dimensões superiores ou transformadas, nas quais os padrões complexos tornam-se mais lineares ou mais fáceis de separar. Em termos práticos, **os neurônios das camadas ocultas** detectam *características* ou *combinações de características* dos dados, e a camada de saída então produz o resultado desejado (por exemplo, uma classificação, uma predição numérica ou, no nosso caso, ações de controle). Essa habilidade de extrair representações úteis e aproximar praticamente qualquer função mensurável (segundo o Teorema da Aproximação Universal) faz das RNAs ferramentas poderosas para inúmeras tarefas em visão computacional, reconhecimento de padrões, robótica, etc.[16][17].

No regime de **aprendizado supervisionado**, a rede neural é treinada fornecendo-se **exemplos rotulados**, ou seja, pares de entrada e saída desejada (também chamados *padrões* e *target*). O algoritmo de treinamento ajusta os pesos da rede de modo que, para cada entrada, a saída produzida se aproxime da saída desejada. Isso é normalmente feito minimizando uma **função de perda** (erro) que mede a discrepância entre a predição da rede e o valor alvo fornecido[18]. O método mais difundido para treinar MLPs é o **backpropagation** juntamente com **otimização por gradiente descendente**, onde o gradiente do erro em relação a cada peso é computado de forma eficiente propagando o erro da saída para trás pelas camadas[19][20]. Em suma, no aprendizado supervisionado a rede aprende com *correção de erros*: sabe-se o que é “certo ou errado” para cada exemplo, e os pesos são ajustados para diminuir o erro a cada iteração.

Por exemplo, se quiséssemos treinar uma rede neural para estacionar um carro de forma supervisionada, precisaríamos de um conjunto de **trajetórias exemplo** ou ações ideais predefinidas – o que não é trivial de obter. Em vez disso, no nosso caso

optamos por uma técnica diferente (evolutiva) onde não fornecemos à rede "como fazer", mas apenas o **critério de sucesso**. Ainda assim, entender o paradigma supervisionado ajuda: muitos problemas clássicos de redes neurais (classificação de imagens, predição de séries, etc.) se enquadram nele.

No **aprendizado supervisionado**, como sumariza Didática Tech (2021), *"o modelo aprende a partir de resultados pré-definidos, utilizando os valores passados da variável target para aprender quais devem ser seus resultados de saída... há uma referência daquilo que está certo e errado"*[21][22]. Essa referência permite medir de forma objetiva a performance da rede durante o treino e guiar a otimização. No treinamento evolutivo do SelfParking, por outro lado, **não teremos um "target" expresso por tempo** (o caminho ou ação corretos a cada instante). Em vez disso, aplicaremos um conceito semelhante à aprendizagem por reforço: o sistema sabe avaliar *após a manobra* se estacionou bem ou mal (através de uma recompensa ou *fitness*), mas não sabe em cada passo qual deveria ser a ação ótima. Por isso, nosso treinamento não será supervisionado clássico, mas ainda assim a rede será ajustada gradualmente visando maximizar um sinal escalar de desempenho.

2.2.2 Aprendizado Não Supervisionado

Em contrapartida ao supervisionado, o **aprendizado não supervisionado** ocorre quando **não há valores de saída desejados** fornecidos ao algoritmo. Ou seja, o sistema recebe apenas entradas e deve descobrir alguma estrutura ou padrão inerente a esses dados por si só[23]. Tarefas típicas de aprendizado não supervisionado incluem **clustering** (agrupamento de dados em categorias desconhecidas), **redução de dimensionalidade** (encontrar representações mais compactas dos dados, como em PCA ou autoencoders) e detecção de anomalias. Por exemplo, se apresentarmos a uma rede um conjunto de imagens sem indicar o que elas representam, o algoritmo pode tentar agrupar imagens similares ou aprender características latentes que comprimem a informação relevante.

No contexto de redes neurais, existem modelos não supervisionados como **autoencoders**, que aprendem a reproduzir seus próprios dados de entrada na saída através de um código interno (a camada escondida menor) – assim aprendendo representações úteis sem rótulos externos. Outro exemplo são **Redes de Kohonen (Mapas Auto-Organizáveis)**, que efetuam clustering competitivo. Conforme explicado por Didática Tech, *"no aprendizado não supervisionado não existem resultados pré-definidos para o modelo utilizar como referência... você apresenta um conjunto de dados e pede para o modelo separar os dados em categorias com base em semelhanças, sem especificar critérios"*[24][25]. Ou seja, o algoritmo tem que **inferir alguma organização interna dos dados sem feedback explícito do que procurar**.

Na problemática do SelfParking, não empregamos diretamente aprendizado não supervisionado, pois temos um critério explícito de sucesso (estacionar ou não, medir colisões, distâncias, etc.). Entretanto, vale mencionar que existe a

possibilidade de treinamento **semissupervisionado ou auto-supervisionado** em controle autônomo, como por exemplo aprender uma representação do ambiente sem rótulos e depois usar essa representação para facilitar o aprendizado de controle. No escopo deste trabalho, a rede neural aprende de forma **orientada por um sinal de recompensa**, não havendo um rótulo de ação correta em cada passo (não supervisionado estrito), mas também não sendo totalmente sem critério (há a recompensa ao final). Esse tipo de aprendizado geralmente é categorizado como uma terceira modalidade: **aprendizado por reforço**, que combina elementos de supervisão (existe feedback de qualidade) com não supervisão (não existe o alvo específico de cada entrada).

Em resumo, **redes neurais artificiais** constituem o núcleo do controlador do nosso sistema. Elas fornecem um formalismo flexível capaz de aproximar a função de controle (das percepções do carro aos comandos de atuador) sem que precisemos definir manualmente uma fórmula de controle. No entanto, redes neurais precisam ser **treinadas**. No presente trabalho, optou-se por não utilizar um algoritmo de treinamento supervisionado tradicional (pois não há dataset de entradas/saídas corretas para o ato de estacionar) nem um algoritmo de backpropagation via reforço (como o policy gradient), mas sim um **algoritmo genético** para encontrar os pesos da rede que maximizam a recompensa cumulativa de estacionamento bem-sucedido. Essa abordagem insere-se dentro de um campo mais amplo conhecido como **neuroevolução**, que será descrito adiante.

2.3 Algoritmos Genéticos

Algoritmos Genéticos (AG) são técnicas de busca e otimização inspiradas nos processos da evolução natural e da genética. Propostos originalmente por John Holland na década de 1960, os AGs simulam a *seleção natural* para encontrar soluções aproximadas para problemas complexos de otimização[26]. A ideia central é representar cada possível solução de um problema como um indivíduo em uma **população**, análogo a um organismo com um genoma, e então aplicar operadores que mimetizam a reprodução e mutação biológicas para evoluir essa população ao longo de diversas **gerações**.

Em um algoritmo genético típico, inicialmente gera-se uma população de soluções **aleatórias** ou heurísticas. Cada indivíduo (solução candidata) é codificado de forma adequada – por exemplo, como uma sequência de bits, números ou qualquer estrutura (o “cromossomo” do indivíduo) – e avaliado por meio de uma **função de aptidão (fitness)** que indica quão boa é aquela solução no contexto do problema. No nosso caso, um indivíduo representará um conjunto de pesos para a rede neural controladora do carro, e a aptidão poderá ser medida de acordo com o quão bem o carro estaciona usando esses pesos.

A seguir, ocorrem iterações com os seguintes **princípios fundamentais** dos AGs[27]:

- **Seleção:** Escolhem-se os indivíduos mais aptos da população atual para contribuir para a próxima geração. A ideia é que soluções melhores tenham mais chance de transmitir suas características (genes) adiante. Existem vários métodos de seleção, como *roleta viciada* (proporcional ao fitness), *torneio* (competição aleatória entre alguns indivíduos) etc.
- **Cruzamento (Crossover):** Dois indivíduos parentais são combinados para produzir novos indivíduos (filhos). Esse operador mistura os "genes" dos pais na esperança de que a combinação herde as boas características de ambos. Por exemplo, se as soluções forem codificadas em vetor, pode-se escolher um ponto de corte e trocar segmentos entre dois pais, gerando dois filhos com partes do cromossomo de cada um.
- **Mutação:** Após o cruzamento, aplica-se uma pequena alteração aleatória em alguns indivíduos (por exemplo, invertendo bits ou somando um ruído pequeno em genes numéricos). A mutação serve para introduzir diversidade genética, evitando convergência prematura e possibilitando explorar novas regiões do espaço de soluções.
- **Elitismo:** É comum incluir também um mecanismo de elitismo, onde alguns dos melhores indivíduos da geração atual são copiados intactos para a próxima geração, garantindo que as soluções de alta qualidade não se percam por azar estocástico nas operações de crossover e mutação.

Esses passos caracterizam um **ciclo evolutivo**: seleção dos mais aptos, reprodução (crossover) gerando descendentes, e mutação adicionando variabilidade[28][27]. Cada tal ciclo produz uma nova geração de soluções. Em geral, espera-se que a aptidão média e principalmente a aptidão do melhor indivíduo aumentem ou ao menos não piorem de geração em geração, à medida que características vantajosas se propagam e refinam. O processo é repetido até algum critério de parada, como atingir um número máximo de gerações ou alcançar um nível de desempenho satisfatório.

A analogia biológica pode ser exemplificada: imagine um problema de otimização onde queremos maximizar uma certa função. Os indivíduos são soluções codificadas, análogas a organismos. A função de aptidão é como o "ambiente" que determina quão bem adaptado cada organismo está – organismos (soluções) com valores altos de aptidão sobrevivem e se reproduzem mais. O crossover é similar à reprodução sexuada, combinando material genético de pais para formar filhos, enquanto a mutação corresponde a erros aleatórios na cópia genética que podem introduzir novas características. Assim, **ao longo de muitas gerações, a população tende a evoluir para regiões cada vez melhores do espaço de soluções**, eventualmente encontrando um ótimo global ou local que seja aceitável[29][30].

Formalmente, algorítmos genéticos pertencem à classe de **algoritmos evolucionários** e **metaheurísticas**, e têm como grande vantagem a habilidade de procurar soluções em espaços muito complexos ou descontínuos onde métodos

analíticos ou de gradiente falhariam. Eles **não garantem** achar a solução ótima global, mas frequentemente encontram soluções boas o suficiente em tempo viável, especialmente para problemas NP-difíceis ou com múltiplos picos locais.

No contexto deste trabalho, usamos um algoritmo genético para otimizar os parâmetros (pesos sinápticos) de uma rede neural. Portanto, cada indivíduo no AG corresponde a um conjunto completo de pesos da rede. Como esses pesos podem ser representados como uma longa lista de números reais, adotamos uma codificação adequada (vetor de real ou eventualmente binário com conversão, se desejássemos simplificar). Nossa função de aptidão será baseada no desempenho de estacionamento do carro simulado quando controlado pela rede neural com aqueles pesos. Ou seja, para **avaliar um indivíduo**, rodamos uma simulação do carro estacionando e medimos a recompensa total obtida (considerando fatores como alcançar a vaga, colisões, tempo, etc.). Esse valor numérico de recompensa será tratado como o *fitness* a ser maximizado pelo AG.

Alguns pontos importantes a destacar sobre o AG empregado:

- Será utilizada uma **seleção por torneio** (conforme implementado, com torneios de tamanho *k* específico). Nesse método, escolhem-se aleatoriamente *k* indivíduos da população e o de maior aptidão entre eles é selecionado para reprodução[31]. Esse processo se repete até ter selecionado suficientes pais. A vantagem é ajustar a pressão seletiva pelo tamanho do torneio: torneios maiores dão mais chance dos melhores vencerem sempre.
- O operador de **crossover** será, por simplicidade, o *crossover* de um ponto (single-point crossover) aplicado sobre a sequência de pesos representando dois pais. Como nossos cromossomos são vetores de números, isso significa pegar uma posição de corte aleatória e trocar as porções de pesos antes e depois desse ponto entre dois pais, gerando dois novos cromossomos filhos[32]. Poderíamos também usar crossover *aritmético* (média de pesos, etc.), mas optou-se pelo simples para preservar interpretabilidade do "material genético".
- A **mutação** consistirá em perturbar alguns pesos com pequena probabilidade. Por exemplo, definir uma taxa de mutação (p.ex. 5% dos genes sofrendo mutação por geração) e, para cada peso escolhido, somar um pequeno valor aleatório gaussiano com média zero ou, em codificação binária, inverter bits. A mutação garante que o algoritmo possa explorar soluções que não estavam presentes inicialmente, ajudando a escapar de ótimos locais.
- O **elitismo** será aplicado conservando os melhores *E* indivíduos de cada geração diretamente na próxima (sem modificações). Isso assegura que a qualidade da melhor solução não decresça de uma geração para outra, pois o melhor anterior estará sempre presente para competir com novos filhos. No

nosso caso, adotou-se um elitismo de tamanho 6 (preservando os 6 indivíduos de maior fitness), conforme parâmetro que será justificado posteriormente.

Assim, o AG funciona como um **processo adaptativo iterativo** que combina busca *exploratória* (via mutações e recombinações inéditas) e *exploitativa* (via seleção preferencial dos melhores existentes). Ao longo das gerações, espera-se evoluir **controladores progressivamente melhores** para a tarefa de estacionar.

Resumindo de forma conceitual, citando uma definição clara: *"Algoritmos Genéticos são uma forma de busca e otimização que imita a evolução natural. Eles operam sobre uma população de soluções candidatas, aplicando seleção, cruzamento e mutação para evoluir soluções cada vez melhores"*[33][34]. A teoria por trás sustenta que, através da **sobrevivência do mais apto** e da **recombinação de partes boas de diferentes soluções**, um AG pode encontrar soluções próximas do ótimo em problemas onde métodos determinísticos falham ou são muito custosos.

Neste ponto convém reforçar: **por que um AG para treinar a rede neural?** A principal razão é que *não sabemos diretamente como calcular o gradiente* do desempenho em relação aos pesos da rede, pois a relação entre pesos e sucesso em estacionar é complexa e não diferenciável globalmente (cheia de platôs e descontinuidades quando há colisões, por exemplo). Além disso, o AG não requer que o problema seja diferenciável nem convexO – apenas requer que possamos avaliar a qualidade de cada solução. Isso o torna muito adequado para nosso caso de simulação, em que podemos simplesmente executar a simulação e obter um escalar de recompensa. A desvantagem potencial é o custo computacional: avaliar muitos indivíduos por geração pode ser lento. Contudo, com simulação rápida e paralelismo (se disponível), isso se torna manejável. Em troca desse custo, ganhamos a possibilidade de escapar de mínimos locais e potencialmente achar soluções inovadoras, pois o AG realiza uma busca global mais abrangente, **explorando simultaneamente múltiplos pontos no espaço de soluções** ao contrário de um gradiente local que ajustaria uma só solução incrementalmente.

2.4 Integração de Algoritmo Genético e Rede Neural

A combinação de algoritmos genéticos com redes neurais resulta em um paradigma de treinamento conhecido como **Neuroevolução**. Nessa abordagem, algoritmos evolutivos (como AG, mas também Estratégias Evolutivas ou Programação Genética) são utilizados para otimizar parâmetros de redes neurais – tipicamente os pesos sinápticos, mas possivelmente também a topologia da rede. O intuito é unir o melhor dos dois mundos: a **capacidade de generalização e aprendizado** das redes neurais com o **poder de busca global** e exploratório dos algoritmos evolutivos[1].

Existem diversas formas de se integrar AGs e RNAs. A mais direta, que é a adotada neste trabalho, é **codificar todos os pesos da rede em um cromossomo** e aplicar o AG para evoluir essa codificação até que a rede atinja um desempenho satisfatório. Assim, cada indivíduo do GA corresponde a uma rede neural inteira. Esse método

evolui apenas os pesos (mantendo a arquitetura fixa pré-definida). Alternativamente, existem abordagens de **neuroevolução estrutural** (como o algoritmo NEAT de Stanley) que começam com redes pequenas e vão também adicionando neurônios e conexões via evolução – mas isso aumenta bastante a complexidade e não foi o foco aqui.

A motivação para usar AG em vez dos métodos de treinamento padrão de redes (como backpropagation) vem de algumas vantagens potenciais apontadas na literatura:

- **Resistência a mínimos locais:** Algoritmos genéticos, sendo baseados em busca estocástica global, não ficam presos facilmente em mínimos locais ruins; a recombinação e mutação podem gerar saltos para fora de vales de soluções subótimas. Em problemas altamente não lineares, isso é um atrativo em comparação ao gradiente, que pode parar em ótimos locais.
- **Otimizam diretamente a medida de desempenho desejada:** No nosso caso, o AG tenta maximizar a própria recompensa acumulada na tarefa de estacionamento. Não é necessário definir gradientes ou ter funções diferenciáveis; basta simular e obter o desempenho. Isso evita a necessidade de diferenciar a função objetivo, algo difícil quando a "função" é um processo simulado complexo.
- **Flexibilidade de design e de restrições:** Podemos incorporar restrições ou múltiplos objetivos no fitness de maneira relativamente simples (por exemplo, penalizar fortemente colisões, ou incorporar critérios como tempo e conforto na função de aptidão). Algoritmos genéticos podem lidar com funções objetivo não suaves ou até descontínuas, o que é complexo para métodos de gradiente.
- **Integração de idéias biológicas**: A neuroevolução também espelha um pouco a ideia de *aprendizado evolutivo* observada na natureza – no nosso caso, estamos "evoluindo" cérebros artificiais (redes neurais) para realizar uma tarefa.

Por outro lado, é importante reconhecer que o AG substitui o papel do algoritmo de aprendizado tradicional e isso pode vir com custo computacional. Enquanto o backpropagation utiliza informação de gradiente para seguir diretamente uma direção de melhoria (descida de gradiente), o AG faz *amostragem* de muitas soluções e seleção, o que geralmente demanda mais avaliações para alcançar boa performance. Entretanto, em alguns casos, a **hibridização** das abordagens é possível: por exemplo, usar um AG para encontrar bons pesos iniciais ou estruturas de rede, e depois usar backpropagation fino para refinar (técnica às vezes chamada de *hybrid learning*). Neste trabalho, focamos somente no AG puro para demostrar a viabilidade sem ajuste por gradiente.

Do ponto de vista conceitual, a integração GA+NN constitui um **sistema de aprendizado híbrido**. Como colocado em um trabalho de revisão: *"Combinando a*

otimização adaptativa dos algoritmos genéticos com a detecção de padrões e modelagem preditiva das redes neurais artificiais cria-se um sistema híbrido que utiliza as forças de ambas as abordagens"[1]. A rede neural sozinha poderia ser treinada por gradiente, mas teria dificuldades sem um conjunto de dados rotulado ou sem um bom algoritmo de exploração. O algoritmo genético sozinho poderia otimizar uma sequência de ações, mas não generalizaria bem para situações diferentes sem um modelo paramétrico. **Juntos**, o AG descobre os parâmetros da rede (um modelo paramétrico generalizador) que melhor realizam a tarefa, e a rede em si, por ter neurônios com funções de ativação não lineares, consegue representar políticas complexas de controle.

Há evidências na literatura de sucesso dessa integração em problemas relacionados. Por exemplo, Feng e Zhu (2020) exploraram geneticamente os pesos de uma rede neural para um carro autônomo de seguimento de trajetória, argumentando que os GA podem substituir a regulação por gradiente tradicional e alcançar aprendizado mais amostral-eficiente em certas tarefas[35][36]. Saez et al. aplicaram GA para otimizar políticas de direção em carros de corrida, minimizando tempo de volta, e obtiveram desempenho competitivo com métodos de otimização clássicos[37][38]. Essas pesquisas reforçam que **algoritmos genéticos podem treinar redes neurais de controle** de veículos de forma eficaz, sobretudo em ambientes de simulação onde é prático avaliar milhares de controladores candidatos.

No nosso caso específico – estacionamento autônomo – a estratégia GA+NN significa: cada ciclo evolutivo testará vários controladores (redes) em estacionar o carro; os melhores controladores "cruzam" entre si (mix de seus pesos) para gerar novos controladores, possivelmente combinando características boas (por exemplo, um que é bom em alinhar com a vaga com outro que é bom em evitar colisão poderá gerar um filho que faça ambos bem). Ao inserir *penalidades* e *bonificações* adequadas no cálculo da aptidão (veremos no capítulo de metodologia), orientamos a evolução para soluções que respeitem todas as condições desejadas.

Em síntese, a **integração de AG e RN** permite **otimizar controladores neurais de forma evolutiva**. Essa combinação visa capitalizar os méritos únicos de cada método para obter soluções melhores do que as obtidas isoladamente[39]. Conforme mencionado: *"Os algoritmos genéticos possuem comprovada habilidade em otimizar soluções para problemas complicados. Uma tendência crescente conecta essa capacidade de otimização com procedimentos de aprendizado de máquina, integrando algoritmos genéticos e redes neurais artificiais para manejar problemas multifacetados difíceis de tratar analiticamente"*[40][1]. O SelfParking se enquadra exatamente nesse contexto, buscando uma solução para um problema de controle não trivial através de um processo de evolução artificial de "redes neurais motoras".

3. Metodologia

Nesta seção é descrita a metodologia empregada no desenvolvimento e experimentação do sistema SelfParking. Abordam-se a construção do **ambiente de simulação**, a configuração e implementação da **arquitetura do sistema** (agente veicular, sensores, rede neural de controle e algoritmo genético), bem como os detalhes da **função de recompensa** e métricas utilizadas para avaliar o desempenho de estacionamento. Também são explicados os procedimentos de treinamento evolutivo, incluindo parâmetros adotados para a execução do algoritmo genético (tamanho da população, número de gerações, taxas de crossover e mutação, elitismo etc.) e os critérios de parada. Por fim, descreve-se a estratégia de **validação** utilizada para testar o controlador treinado em situações variadas, introduzindo aleatoriedades (ruídos) nas condições iniciais para avaliar a capacidade de generalização do sistema.

3.1 Ambiente de Simulação

Para treinar e testar o agente de estacionamento de forma segura e repetitiva, utilizou-se um **ambiente de simulação virtual** construído sobre a plataforma **Unity3D**. O Unity foi escolhido por oferecer um motor de física robusto, facilidade de visualização em 3D/2D, e por dispor do pacote **ML-Agents** que facilita a integração de agentes de aprendizado por reforço dentro de cenas simuladas. Embora neste projeto específico o algoritmo de aprendizado (AG) não seja um dos fornecidos pelo ML-Agents, ainda assim aproveitou-se a estrutura de agentes e sensores do Unity.

O ambiente simulado reproduz um **estacionamento planar** visto de cima (top-down), incluindo vagas demarcadas, obstáculos e veículos estacionados de forma similar ao mundo real. Foi modelada uma área retangular com solo plano (asfalto) e delimitada por paredes nas bordas (para simular limites do estacionamento). Dentro dessa área, há **vagas de estacionamento** delineadas com marcações amarelas. Algumas vagas podem estar ocupadas por carros estacionados (veículos obstáculo), de modo a deixar apenas uma vaga livre para o agente estacionar – replicando a situação comum de estacionar entre dois carros. A posição e ocupação das vagas podem ser configuradas no início de cada episódio para variar as condições.

O agente principal é representado por um **modelo simples de carro** (um objeto 3D com colisor e dinâmica), dotado de um controlador de física para translacional e rotação. Foram considerados dois graus de liberdade principais de controle: aceleração (para frente e ré, incluindo freio) e esterçamento (ângulo do volante). Assim, o carro pode se deslocar para frente ou trás e girar, permitindo manobras de baliza ou ajuste fino de posicionamento.

Uma decisão importante foi simplificar a simulação para ser *quase cinemática*. Isto é, embora use o motor físico para colisões e dinâmica básica, não simulamos motor, marcha ou atrito de forma complexa; privilegia-se uma dinâmica simplificada onde os comandos de aceleração e direção resultam em movimento controlado e

idealizado. Essa simplificação serve para reduzir a complexidade do aprendizado, focando nos aspectos essenciais da tarefa (geometria da manobra) sem introduzir dificuldades mecânicas excessivas. Ainda assim, ruídos podem ser adicionados e a física do Unity confere realismo suficiente (por exemplo, o carro tem inércia e não para instantaneamente).

Sensores: No ambiente virtual, equipamos o veículo agente com sensores que fornecem leituras necessárias para tomada de decisão autônoma. Como não se utilizou processamento de visão (câmeras) neste trabalho, optamos por sensores simulados de distância do tipo **raycast** (raios). Foram acoplados ao carro diversos *Raycast Sensors*, similares aos componentes `RayPerceptionSensorComponent3D` do ML-Agents[41], que emitem raios em direções específicas a partir do veículo e detectam colisão com certos objetos etiquetados (por exemplo, paredes, carros, vaga-alvo). Esses sensores fornecem distâncias normalizadas até o obstáculo mais próximo em cada direção. No total, definimos **Nove raios** distribuídos ao redor do veículo cobrindo 180 graus frontais (em leque) e alguns traseiros, de forma a prover percepção de obstáculos nos lados frontal, diagonal e traseiro do carro. Cada leitura de distância é usada como entrada para a rede neural de controle. Na prática, os sensores se assemelham a sensores ultrassônicos ou LiDAR simples que medem proximidade de paredes e outros carros. *Figura 1* ilustra o cenário do estacionamento simulado e a configuração dos sensores em torno do veículo.

(Figura 1 – captura de tela do ambiente Unity em visão superior, mostrando o carro verde (agente) e vagas de estacionamento demarcadas; ilustrar com setas/raycasts as direções dos sensores de distância do agente)[42]

Adicionalmente aos raios de distância, o agente obtém algumas outras informações do ambiente para auxiliar o aprendizado: - A **posição relativa da vaga alvo** em relação ao carro, expressa em coordenadas no referencial do veículo (por exemplo, fornecendo Δx e Δz da vaga em coordenadas locais do carro). Isso dá ao agente uma noção de onde está a vaga que deve ocupar. - A **orientação desejada** do carro quando estacionado (tipicamente, paralela às vagas), também fornecida como vetor ou ângulo relativo, para ajudar o agente a alinhar o veículo corretamente. - A **velocidade atual** do carro, que pode ser útil para o agente saber se está se movendo e ajustar comandos proporcionalmente. Usamos a projeção da velocidade linear no referencial do carro (componentes forward e lateral da velocidade) como entradas adicionais[43].

Essas observações adicionais somam-se às leituras dos sensores de distância, formando o vetor de **entrada da rede neural**. Todas as observações são normalizadas em faixa adequada (por exemplo, distâncias divididas pelo alcance máximo, velocidade escalonada pelo valor máximo possível, posições relativas escaladas pelo tamanho do estacionamento, etc.) para facilitar o processamento pela rede neural.

Como base de comparação, a Unity ML-Agents tipicamente integra esses sensores e observações de forma transparente: *"os valores dos sensores de raycast são automaticamente adicionados na observação a cada passo"*[44][45]. Aqui implementamos manualmente lógica semelhante: a cada iteração de controle, reunimos as distâncias detectadas e outros parâmetros e repassamos ao controlador neural.

Episódios de simulação: O treinamento é conduzido em episódios discretos. No início de cada episódio, o agente (carro) é posicionado em uma localização inicial aleatória no estacionamento, com orientação também variando dentro de certo intervalo. Por exemplo, o carro pode começar alguns metros afastado da vaga alvo, em ângulo deslocado. Também a ocupação das vagas pode ser alterada: outros carros podem estar estacionados em vagas adjacentes ou não, para diversificar cenários (como vagas paralelas ou oblíquas). Essa *aleatorização inicial* visa tornar o aprendizado mais robusto, forçando o agente a aprender uma estratégia geral de estacionamento e não apenas memorizar um trajeto fixo a partir de uma posição específica[46].

Um episódio termina quando o carro **conclui o estacionamento** (critério de sucesso) ou quando atinge alguma condição de falha ou tempo máximo. Definiu-se que um estacionamento é bem-sucedido se a posição do carro fica dentro dos limites da vaga designada (com tolerâncias de distância) e sua orientação final está alinhada adequadamente (dentro de certo desvio angular permitido), permanecendo parado (velocidade próxima de zero). Ao detectar essas condições, o episódio é encerrado com sucesso. Condições de falha incluem: - **Colisão**: se o carro colidir com um obstáculo (outro veículo estacionado, parede ou delineador de vaga) antes de estacionar, o episódio termina imediatamente com falha. - **Tempo esgotado**: cada episódio tem duração máxima (em passos de simulação). Se esse limite for atingido sem estacionar, considera-se falha (ou simplesmente termina o episódio e avalia-se parcial). - **Saída dos limites**: se o carro se afasta demais da região relevante (por exemplo, sai da área do estacionamento), interrompe-se o episódio por segurança.

Essas definições serão importantes para computar a função de recompensa. Mas em termos de simulação, garantem que não fiquemos com episódios travados indefinidamente – o carro tem uma janela de tempo para tentar estacionar e, se não conseguir, resetamos para tentar novamente (como aconteceria se falhasse).

Em suma, o ambiente de simulação provê um **laboratório controlado** para o nosso agente evoluir: é suficientemente fiel para emular os desafios geométricos e físicos de estacionar (espaços limitados, necessidade de manobras de ré, evitar colisão), e ao mesmo tempo personalizável e repetível para o treinamento massivo necessário pelo algoritmo genético. A utilização do Unity facilitou incorporar sensores visuais (raycasts) e controlar episódios de forma programática (via scripts C# associados ao agente e ao gerenciador de simulação). Cabe ressaltar que não se utilizou diretamente os algoritmos de treinamento do ML-Agents (como PPO ou SAC), mas a

infraestrutura de agentes e ambiente serviu de base para implementar nosso próprio laço evolutivo de treinamento dentro do Unity.

3.2 Arquitetura do Sistema *SelfParking*

A figura a seguir apresenta a arquitetura geral do sistema desenvolvido, indicando seus principais componentes e o fluxo de informações entre eles.

(Figura 2 – Diagrama em blocos da arquitetura SelfParking: incluir o bloco do agente (carro) com sensores alimentando a rede neural, a rede neural gerando comandos para o veículo, e um bloco externo de Algoritmo Genético ajustando os pesos da rede a cada geração. Mostrar também o ambiente simulado fechando o loop, onde o veículo interage e produz uma recompensa.)

Conforme ilustrado, a arquitetura *SelfParking* pode ser dividida em quatro elementos principais: **Agente (veículo) e Sensores**, **Controlador Neural**, **Avaliador de Desempenho (Recompensa)** e **Algoritmo Genético**. A seguir, cada um desses componentes é detalhado.

3.2.1 Agente (Veículo) e Sensores

O **agente** é o carro simulado, dotado de atuadores e sensores virtuais. No Unity, ele foi implementado como um objeto com um componente de script (chamado `CarAgent` por exemplo) que controla a interface com o ambiente. Esse script possui métodos para: - **Receber ações** (comandos de aceleração e direção) e aplicá-las ao veículo (ajustando torque do motor e ângulo das rodas dianteiras, no caso de carro com tração dianteira e esterço nas rodas frontais). - **Coletar observações** do estado atual: leituras dos sensores de distância, velocidade do carro e pose relativa da vaga alvo, conforme descrito na seção anterior. - **Detectar eventos** de interesse, como colisões (através de callbacks de colisão do Unity, e.g., `OnCollisionEnter`), ou entrada na vaga (através de colisores gatilho posicionados nas vagas)[47][48].

Os **sensores** do agente consistem em um conjunto de *raycasts* configurados via script, disparados a cada atualização de observações. No momento de coletar as observações (método equivalente a `CollectObservations` do ML-Agents ou chamado diretamente pelo nosso laço de controle), esses raios retornam distâncias normalizadas ou um valor máximo se nada for atingido dentro do alcance. Também são verificadas tags dos objetos atingidos para distinguir se o obstáculo é um carro estacionado, uma parede, etc., caso fosse necessário tratamento diferenciado (neste trabalho, usamos principalmente a distância pura e eventualmente marcamos diferentemente a vaga alvo para um sensor específico detectá-la, como descrito abaixo).

Adicionalmente aos **n sensores de distância**, foi usado um artifício para indicar ao agente onde está a vaga livre que ele deve ocupar. Implementou-se uma espécie de "sensor virtual" que aponta para o **centro da vaga alvo**. Isso foi feito de duas formas redundantes: - Um **raycast direcionado à vaga**: alguns experimentos incluíram um

sensor raycast específico apontado na direção aproximada da vaga alvo, que detecta quando o carro está alinhado com ela. Esse sensor detecta um objeto “alvo” posicionado no centro da vaga (representado por um pequeno cubo invisível com uma tag especial). Quando o raio o “vê”, retorna a distância até ele; caso contrário retorna a distância máxima. Esse método fornece tanto a distância frontal à vaga quanto indica se está na direção certa. - Um **vetor posicional relativo**: conforme mencionado, calculamos a posição do centro da vaga no referencial do carro (fazendo transformação de coordenadas). Esse vetor é então incluído como três observações (deslocamentos X, Z e possivelmente orientação relativa ou dot product com frente do carro)[49][50]. Isso informa ao agente “a vaga está 5m à frente e 2m à direita, por exemplo”. Junto com a distância dos sensores, esse vetor ajuda o agente a se guiar.

Essas estratégias garantem que o agente tenha alguma noção do objetivo final (vaga) além de apenas evitar obstáculos. Isso é importante para reduzir ambiguidade: caso contrário, o agente poderia aprender apenas a não bater e ficar rodando indefinidamente sem saber onde estacionar. Ao dar a ele um indicador explícito da vaga, direcionamos o comportamento.

Atuadores (Ações): As ações que o agente executa são representadas por um vetor contínuo de 2 dimensões: 1. **Steering** (direção do volante): valor contínuo normalizado entre -1 e 1, correspondente ao ângulo máximo de esterçamento para esquerda ou direita. Por exemplo, -1 = virar totalmente à esquerda, 0 = volante centralizado, 1 = totalmente à direita. Foi considerado um ângulo máximo de cerca de 30 graus para cada lado, em linha com parâmetros do veículo simulado[51]. 2. **Throttle/Brake** (aceleração): valor contínuo entre -1 e 1, onde valores positivos representam acelerar para frente, valores negativos aceleram para trás (ré), e valor 0 significa soltar o acelerador (o que, dependendo da física, age como freio motor). Implementou-se também que se o valor de throttle for zero e o carro estiver em movimento, aciona-se um **freio** proporcional para fazê-lo parar gradualmente[52][53]. Em suma, essa única saída controla tanto acelerador quanto ré e freio.

Em algumas tentativas iniciais, considerou-se separar freio em outra ação (como visto em um exemplo da literatura com 3 sinais: direção, aceleração e freio)[54], porém verificou-se que para simplificar o espaço de ação e evitar redundância, um controle unificado de throttle positivo/negativo seria suficiente: throttle negativo implica ir de ré, e se o carro precisar frear em meio a uma aceleração positiva, o agente pode simplesmente reduzir rapidamente o valor para zero ou negativo. Então optamos por **2 ações contínuas** em vez de 3, o que também reduz a dimensão do problema para a rede neural.

Laço de controle: Durante cada passo de simulação (o Unity permite definir a frequência de decisão), o agente lê seus sensores (observações), alimenta a rede neural (que calculará as ações) e aplica essas ações ao Rigidbody do veículo. O

passo de física do Unity então evolui o estado do veículo e do ambiente, detectando colisões etc., e o ciclo se repete. Utilizou-se um *Time Scale* acelerado (por exemplo, 2x) durante treinamento para agilizar as iterações, mantendo ainda estabilidade. Entre episódios, como mencionado, a posição do agente e a ocupação das vagas são redefinidas.

Resumindo, o Agente e seus sensores constituem a **interface entre o controlador e o ambiente**. Ele fornece à rede neural uma representação compacta do estado (distâncias, vetores relativos, velocidades) e executa os comandos contínuos resultantes. Todo o restante do trabalho – a inteligência de estacionar – reside nos **parâmetros internos da rede neural**, os quais serão ajustados pelo algoritmo genético conforme descrito a seguir.

3.2.2 Controlador Neural (Rede MLP)

O **controlador neural** do SelfParking é uma **Rede Neural Artificial** do tipo **Perceptron Multicamadas (MLP) feedforward**, implementada e executada em tempo de simulação dentro do Unity (usando código C# customizado para inferência). Essa rede recebe como entrada o vetor de observações do agente a cada instante e produz como saída o vetor de ações contínuas (steering e throttle) a serem aplicadas.

Arquitetura da Rede: Após experimentação preliminar, definiu-se uma arquitetura com **3 camadas ocultas** densas (totalmente conectadas) de tamanhos decrescentes, e uma camada de saída: - Camada de entrada: dimensão igual ao número de observações do agente. Supondo que tenhamos, por exemplo, 9 sensores de distância + 3 componentes de vetor para vaga + 3 de velocidade + possivelmente 1 bias extra = em torno de 16 entradas (valor exato dependendo de sensores finais utilizados). - **1ª camada oculta:** 32 neurônios (com função de ativação não-linear ReLU, por exemplo). - **2ª camada oculta:** 16 neurônios (ReLU). - **3ª camada oculta:** 8 neurônios (ReLU). - Camada de saída: 2 neurônios lineares (correspondendo às ações de direção e aceleração). Aqui foi usada ativação linear ou às vezes tanh para naturalmente limitar entre -1 e 1. No nosso caso, optou-se por ativação **tanh** nos neurônios de saída para garantir que os valores sejam emitidos no intervalo [-1,1] sem saturar abruptamente (o tanh tende a saturar, mas de forma suave). Também poderia ser usada função logística ou escala manual após linear.

Os tamanhos acima foram selecionados de maneira a fornecer capacidade suficiente de representação sem tornar o número de pesos exorbitante. Com essa configuração, o total de pesos (parâmetros livres) da rede é: 16x32 + 32 (bias) + 32x16 + 16 + 16x8 + 8 + 8x2 + 2 = **770 pesos e bias** aproximadamente (valor exato pode variar dependendo do número de entradas final). Esse número de ~770 parâmetros é relativamente modesto, permitindo que o algoritmo genético manipule cromossomos de tamanho <1000, o que é tratável em termos de performance e convergência.

Se a rede fosse muito pequena, talvez não capturasse a complexidade da manobra de estacionamento; se fosse muito grande (milhares de neurônios), o AG teria muita dificuldade de evoluir devido à alta dimensionalidade. Os valores 32-16-8 foram testados empiricamente e mostraram funcionar bem (ver seção de resultados). Vale notar que **camadas ocultas menores** forçam a rede a aprender representações compactas – uma espécie de regularização – podendo favorecer a generalização. Por outro lado, camadas muito pequenas poderiam limitar a política (por exemplo, 8 neurônios apenas talvez não conseguissem modelar bem curvas suaves de aproximação da vaga). O balanço escolhido pareceu adequado.

Funções de Ativação: Como mencionado, utilizou-se a função **ReLU (Rectified Linear Unit)** nas camadas ocultas. A ReLU define $f(x) = \max(0, x)$, introduzindo não linearidade e parcimônia de forma simples e eficiente, além de evitar o problema de gradientes desaparecendo que ocorre com sigmoide em redes profundas (embora nesse trabalho específico não há treinamento via gradiente, ainda assim a ReLU facilita a exploração do espaço de pesos pelo GA, pois produz saídas proporcionais sem saturação em larga faixa). Na camada de saída, a opção foi ativação **tangente hiperbólica (tanh)** para limitar naturalmente as ações. Assim, a rede sempre devolverá valores entre -1 e 1 para cada ação, que então são diretamente interpretados como comandos. Isso elimina a necessidade de truncar ou normalizar as saídas manualmente.

Execução da Rede: Cada vez que o agente precisa decidir uma ação, ocorre uma passagem feedforward pelos neurônios: 1. Os valores de entrada (observações normalizadas) são carregados nos neurônios da camada de entrada. 2. Calcula-se para cada neurônio da primeira camada oculta a soma ponderada das entradas mais um termo bias, e aplica-se ReLU. 3. Propaga-se para a segunda camada oculta, repetindo o cálculo com os outputs da primeira. 4. Mesmo procedimento na terceira camada oculta. 5. Por fim, computa-se os neurônios de saída como combinações lineares da última camada oculta, e aplica-se tanh nesses somatórios para obter as ações.

Tudo isso é implementado por operações matemáticas relativamente simples (somatórios e funções), que rodam muito rapidamente na CPU mesmo para dezenas de neurônios. Portanto, a inferência em tempo real não é um gargalo – Unity lida facilmente com isso, e inclusive se fosse necessário, poder-se-ia usar burst compiler ou Jobs para paralelizar, mas não foi preciso.

A rede neural assim funciona como o **cérebro do agente**, tomando decisões baseadas nas percepções. No início do treinamento, as **pesos da rede são inicializados aleatoriamente** (cada indivíduo do GA começa com um conjunto aleatório de pesos). Essas redes inicialmente não têm comportamento coerente – o carro se moverá aleatoriamente ou ficará girando. Durante o processo evolutivo, os pesos serão ajustados por seleção natural para gerar comportamentos cada vez mais próximos do desejado (manobras de estacionamento bem-sucedidas).

Importante ressaltar que **não há aprendizado online** do tipo backpropagation ocorrendo durante um episódio. Ou seja, enquanto o agente executa um episódio de estacionamento, seus pesos ficam fixos. Somente entre gerações do AG (após avaliar toda a população) é que ocorrem mudanças nos pesos – e essas mudanças são pelas operações genéticas, não por gradiente de erro instantâneo. Em outras palavras, **o aprendizado acontece na escala evolutiva (entre gerações), não na escala de um episódio único**. Isso distingue radicalmente do típico agente de aprendizado por reforço, que ajusta um pouco os pesos após cada episódio via gradiente; aqui, acumulamos desempenho e só no final de toda uma geração aplicamos as operações de reprodução para criar uma nova geração de redes.

Em termos de implementação, a representação do cromossomo (conjunto de pesos) foi mapeada diretamente para arrays de floats na memória. Escreveu-se uma função para **aplicar os pesos** de um cromossomo à rede neural do agente antes de rodar a simulação. Dessa forma, ao avaliar um indivíduo, o sistema pega seus pesos, coloca na estrutura da rede, e então executa o episódio. Após o episódio, registra a recompensa obtida. Depois, o AG decide, com base nas recompensas de todos, quem sobrevive e quem reproduz.

3.2.3 Algoritmo Genético para Otimização de Controle

O **Algoritmo Genético (AG)** implementado é o responsável por treinar (evoluir) o controlador neural descrito acima. Ele opera em um ciclo de gerações, ajustando progressivamente os pesos da rede neural para maximizar a recompensa de estacionamento. A seguir são detalhados os parâmetros e o funcionamento específico do AG no sistema SelfParking.

Codificação dos Indivíduos: Cada indivíduo na população do AG representa **uma configuração completa de pesos** da rede neural (todos os pesos e biases). Usou-se uma codificação direta em vetor de números reais de dimensão $D \approx 770$ (ou o número exato de parâmetros). Ou seja, o cromossomo de um indivíduo é um vetor $W = [w_1, w_2, ..., w_D]$, onde cada w_i corresponde a um peso sináptico ou viés da rede. Esses valores inicialmente são gerados aleatoriamente dentro de um intervalo (por exemplo, $[-1, 1]$ ou com distribuição normal de média 0 e desvio pequeno), semelhante à inicialização aleatória usual de redes neurais. Não foi usada codificação binária por bitstring, preferiu-se real para evitar quantização e facilitar mutações sutis nos valores.

Tamanho da População: Fixou-se uma população de **50 indivíduos** por geração (valor escolhido considerando um compromisso entre diversidade e custo computacional de avaliar todos). Assim, a cada geração, 50 redes neurais diferentes são instanciadas e avaliadas. Esse número mostrou-se suficiente para que o AG explorasse uma gama de estratégias e mantivesse diversidade genética, sem ser excessivamente grande a ponto de inviabilizar o tempo de simulação (avaliar 50 indivíduos por geração é administrável, enquanto populações muito maiores linearly aumentariam o tempo de cada geração).

Avaliação (Função de Aptidão): A função de aptidão do AG está intrinsecamente ligada à **função de recompensa** do problema de estacionamento, que será detalhada na seção 3.3. Em resumo, para cada indivíduo (conjunto de pesos), é executado um episódio completo de simulação do estacionamento. Durante esse episódio, o agente controlado por aquela rede neural recebe recompensas incrementais e penalidades de acordo com seu comportamento (ex: penalidades por colisão, pequenas penalidades por tempo gasto, recompensas por se aproximar da vaga, etc.) e uma grande recompensa final se conseguir estacionar. Ao fim do episódio, a **aptidão** do indivíduo é definida como o total de recompensa acumulada. Alternativamente, pode-se definir aptidão de forma mais direta: por exemplo, aptidão = -distância final até o centro da vaga (quanto menor a distância, melhor, e estacionado perfeito dá distância zero), adicionando um termo de penalização se colidiu. Porém, adotamos a soma de recompensas do próprio esquema de reforço como aptidão, o que em prática equivale a isso pois projetamos a recompensa para refletir esses critérios.

É importante mencionar que para reduzir estocasticidade, optou-se por avaliar **cada indivíduo mais de uma vez em condições diferentes** e usar a média como aptidão. Isso porque, se o indivíduo for avaliado em apenas uma situação, pode haver sorte ou azar (por exemplo, ele nasceu calibrado sem querer para uma posição inicial específica e se sai bem nela, mas iria mal em outras). Para ter uma medida mais robusta, cada indivíduo foi testado em **3 episódios diferentes** na mesma geração, variando levemente a posição inicial ou o layout (via seeds diferentes), e o fitness considerado é a média das recompensas obtidas. Esse número 3 é arbitrário mas aumenta um pouco a confiabilidade da avaliação sem multiplicar demais o tempo. Futuros experimentos poderiam aumentar para 5 ou 10 se necessário. Em implementações com ML-Agents, isso seria análogo a rodar o mesmo policy em múltiplos instâncias paralelas do ambiente e promediar resultados.

Seleção dos Pais: Após avaliar todos os indivíduos da geração, o AG procede a formar a próxima geração. Utilizou-se o método de **Seleção por Torneio** (Torneio de tamanho $K=3$). O procedimento é: - Repetidamente, escolher aleatoriamente 3 indivíduos distintos da população atual. - Dos escolhidos, verificar suas aptidões e selecionar o de maior aptidão como vencedor do torneio. - Esse vencedor é copiado para a pool de pais selecionados.

Esse processo é repetido até termos selecionado o número necessário de pais para reprodução. Como usamos reprodução sexual (dois genitores por cruzamento), precisamos selecionar um número par de pais. Na nossa implementação, para gerar N novos indivíduos (que será igual ao tamanho da população, excetuando elites), selecionamos N pais (na verdade $N/2$ pares). O torneio fornece pressões seletivas controláveis: com $K=3$, a probabilidade do melhor de toda a população ser selecionado em pelo menos um torneio é alta, mas os medianos também têm chance se por sorte caírem em torneios contra indivíduos ainda piores. Isso mantém

diversidade. Além disso, aplicamos **elitismo** separadamente, então não precisamos forçar demais a seleção para preservar bons indivíduos.

Elitismo: Foi implementado elitismo com $E = 6$ indivíduos. Isto é, os 6 melhores indivíduos de cada geração (em termos de aptidão) são automaticamente copiados para a geração seguinte **sem modificação**. Esses indivíduos elitistas ocuparão 6 vagas da população da próxima geração, garantindo que não percamos as melhores soluções encontradas até então. O restante (50 - 6 = 44 indivíduos) será preenchido pelos filhos gerados via reprodução dos pais selecionados. Elitismo garante monotonia não decrescente do melhor fitness ao longo das gerações (o melhor nunca piora), embora possa reduzir diversidade – escolhemos 6 (12% da pop) o que é relativamente conservador para manter também diversidade nos outros 88%.

Cruzamento (Reprodução): Para formar novos indivíduos, aplicou-se o operador de **Cruzamento de Um Ponto (single-point crossover)**. O algoritmo pega dois pais selecionados (vetores de pesos P_1 e P_2), escolhe um índice de corte aleatório c entre 1 e $D-1$ (onde D é tamanho do cromossomo), e então gera dois filhos: - Filho1 recebe os genes $[w_1 ... w_c]$ do Pai1 e $[w_{c+1} ... w_D]$ do Pai2. - Filho2 recebe os genes $[w_1 ... w_c]$ do Pai2 e $[w_{c+1} ... w_D]$ do Pai1.

Esse é o crossover clássico. Exemplo: se $c = 500$ e cada peso for número real, significa que o primeiro 500 pesos do filho vêm do primeiro pai e o restante do segundo pai. Com isso, cada filho carrega aproximadamente metade da "herança" genética de cada progenitor. Este método pressupõe que a ordem dos genes tem certa correlação estrutural (no caso de pesos de rede, eles são ordenados por camadas e conexões, então genes contíguos correspondem a subconjuntos de conexões na rede). É uma suposição razoável – e mesmo que não fosse, o GA tende a se adaptar caso haja epistasia entre genes.

Observação: Poderíamos ter usado *crossover de dois pontos* ou *crossover uniforme*. Testamos também crossover de dois pontos, mas não notamos diferença substancial. O uniform (que escolhe cada gene de pai1 ou pai2 com prob 0.5) pode introduzir muita disruptividade. Então mantivemos um ponto.

Cada par de pais gera assim 2 filhos. Como selecionamos 44 indivíduos para repor (lembrando dos 6 elites), precisamos de 44 novos indivíduos. Logo, selecionamos 22 pares (44 pais) para cruzamento, gerando 44 filhos. Assim completamos a população de 50 da nova geração (6 elites copiados + 44 filhos novos).

Mutação: Depois do cruzamento, cada gene de cada novo indivíduo (exceto os elites, que não passam por modificação) tem uma chance de sofrer **mutação**. Definiu-se uma **taxa de mutação base de 20%** (0,2). Isso significa que, em média, 20% dos pesos serão alterados de forma aleatória após o crossover. Esse valor é relativamente alto comparado ao que se vê em otimização binária clássica (onde costuma-se usar 1% ou menos), mas em otimização contínua de redes neurais

muitos autores usam mutações mais frequentes para fornecer mais variedade, já que o espaço é contínuo e as mutações podem ser pequenas.

A mutação em cada peso consiste em adicionar um pequeno valor aleatório proveniente de uma distribuição Gaussiana com média 0. Adotamos um desvio padrão inicial de 0.05 (5% da amplitude típica). Esse desvio pode ser adaptativo – implementamos uma forma de **mutação adaptativa** que reduz o desvio conforme as gerações avançam, permitindo ajustes mais finos no fim do treinamento. Especificamente, definimos um *decay* linear para o desvio de mutação ao longo de um certo número de gerações, ou alternativamente usamos a estratégia de *1/5 success rule* adaptativa (não implementada totalmente devido à dificuldade de medir "sucesso" definidamente aqui). Mas em testes, uma mutação com desvio fixo também funcionou, então manter simples: $\sigma_{mut} = 0.05$ constante.

Além da mutação gaussiana nos pesos, introduziu-se uma pequena chance de **"imigrantes"** aleatórios – isto é, com uma probabilidade de 5% um novo indivíduo gerado era substituído por um indivíduo totalmente aleatório. Essa técnica de *imigrant replacement* ajuda a introduzir novos genes não dependentes da população atual. Em nossa implementação, parametrizei como "Immigrant Rate = 0.05", ou seja, espera-se ~2 indivíduos por geração serem randômicos. Isso evita convergência prematura ao repovoar com diversidade ocasionalmente.

Por fim, os indivíduos elitistas copiados também podem ou não ser protegidos de mutação. Normalmente, no elitismo puro, você copia sem mutar (como fizemos, os 6 foram carregados intactos). Mas para adicionar leve exploração até mesmo nos top, alguns experimentos aplicam mutação mínima nos elites. Aqui mantivemos elites intactos para garantir aquele piso de qualidade.

Integração AG + Simulação: A implementação prática do AG ocorreu dentro do próprio Unity em modo *training*. Um componente `GAController` orquestrava as gerações: - Inicializa população aleatória. - Para cada indivíduo: - Carrega os pesos na rede neural do agente. - Reseta o ambiente (posiciona carro, define cenário). - Executa a simulação até terminar o episódio (ou limite). - Recebe a recompensa acumulada (fitness). - Repete mais vezes se múltiplas avaliações por indivíduo, calcula média. - Após todos avaliados, guarda o melhor fitness e solução. - Realiza seleção, crossover e mutação para criar nova população. - Itera para a próxima geração.

Imprimiu-se logs a cada geração mostrando: geração número X, melhor aptidão = Y, aptidão média = Z, percentual de sucesso (quantos estacionaram com sucesso) etc., para monitorar a evolução. A interface de console do Unity exibia essas informações. Por exemplo, logs típicos: *"Gen 0 | best = 540.18 | mean = 535.54 | parked = 100%"* (ilustrativo). Nesse caso, significava que já no gen 0 todos estacionaram? Provavelmente log de teste com um cenário trivial. Mas ao longo do treinamento real, viu-se uma melhoria progressiva dessas métricas (ver resultados).

Critérios de Parada: O algoritmo genético foi executado por um número pré-definido de gerações, configurado como **2000 gerações** no experimento principal. Esse número foi escolhido baseado em testes iniciais onde a aptidão parecia convergir antes disso – por volta de 1500 gerações já não havia ganhos significativos. Assim, 2000 gerações serve como teto. Além disso, estabeleceu-se um critério opcional: se o **desempenho ótimo** atingisse o máximo teórico (por exemplo, se o melhor indivíduo conseguir a pontuação máxima possível da recompensa, indicando estacionamento perfeito) e mantivesse isso por várias gerações, poderíamos interromper antes. Na prática, a pontuação máxima possível foi calibrada e raramente atingida exatamente, então rodou-se até as 2000 por garantia.

O diagrama abaixo resume o fluxo do algoritmo genético:

- **Início:** Gerar população inicial aleatória de tamanho N.
- Avaliar cada indivíduo (simular episódio(s) e calcular fitness).
- Computar estatísticas (melhor, média).
- **Seleção:** Usar torneios para selecionar pais.
- **Reprodução:** Cruzar pais para gerar filhos até preencher N - E vagas.
- **Elitismo:** Copiar os E melhores da geração atual para a próxima.
- **Mutação:** Aplicar mutação aos filhos (e possivelmente uma leve nos pais selecionados, se implementado).
- Formar nova geração com elites + filhos mutados.
- **Loop:** Repetir avaliação da nova geração... (até atingir geração final ou convergência).

(Figura 3 – Fluxograma do Algoritmo Genético empregado, mostrando as etapas de avaliação, seleção por torneio, crossover, mutação e elitismo, e o loop de gerações.)[55][56]

Com essa metodologia de treinamento evolutivo definida, garantimos que o sistema SelfParking tenha condições de aprender a estacionar a partir do zero, apenas com o feedback de sucesso ou falha nas tentativas simuladas. Nos próximos tópicos, definiremos formalmente a função de recompensa utilizada e discutiremos as métricas de avaliação e experimentos conduzidos.

3.3 Função de Recompensa e Métricas de Avaliação

Projetar a **função de recompensa** (ou função de aptidão) adequada foi um passo crucial, pois ela guia todo o processo de aprendizado evolutivo. Para que o algoritmo genético evolua comportamentos desejados, a recompensa deve refletir fielmente o que consideramos um estacionamento bem-sucedido, além de fornecer sinais para comportamentos intermediários. Nesta seção, descrevemos as recompensas e penalizações definidas, bem como as métricas utilizadas para avaliar quantitativamente o desempenho do agente.

Objetivos do Comportamento Desejado: Em termos qualitativos, queremos que o agente: - Alcance a vaga designada e pare o veículo dentro dos limites da vaga. - Esteja **centralizado e alinhado** na vaga (não enviesado ou atravessado). - **Não colida** com nenhum obstáculo (carros adjacentes, paredes, etc.) no processo. - Realize a manobra de forma eficiente, ou seja, sem movimentos desnecessários e dentro de um tempo razoável.

Com base nesses objetivos, criamos componentes de recompensa correspondentes:

1. **Recompensa de Sucesso (Jackpot):** Ao estacionar corretamente na vaga, o agente recebe uma recompensa positiva grande, equivalente a completar a tarefa. Foi atribuído, por exemplo, **+1000 pontos** para um estacionamento bem-sucedido. Esse valor alto serve para que qualquer solução que estacione (mesmo que com alguns erros menores) seja claramente distinguível em aptidão de uma solução que não estaciona. Implementacionalmente, detecta-se o sucesso via um *gatilho de vaga*: quando o veículo entra completamente na vaga e para, disparando a condição de finalização do episódio, então se concede essa recompensa[48][57]. Além disso, acrescentou-se um bônus dependente do alinhamento final: verificou-se o ângulo do carro em relação ao eixo da vaga; quanto mais alinhado (próximo de 0° de desvio), maior o bônus dentro de um intervalo (por ex, até +200). Um carro estacionado de ré mas de frente para o muro (ou seja, virado 180° errado) receberia o mínimo desse bônus. Esse mecanismo foi inspirado em Raju (2022), onde eles mencionam dar *"um jackpot e um bônus baseado em quão alinhado o agente está ao estacionar"*[48][57]. No nosso caso:
2. Jackpot base: +1000
3. Bônus de alinhamento: até +200 (proporcional ao cosseno do ângulo de orientação em relação ao desejado, por exemplo).
4. Com isso, estacionar perfeitamente alinhado poderia render até +1200 no final.

5. **Penalidade por Colisão:** Qualquer colisão com objetos relevantes (carro obstáculo, parede, ou sair muito do limite) acarreta uma penalização severa e o fim do episódio. Aplicou-se uma penalidade de **-500 pontos** ao colidir, além de terminar o episódio imediatamente (o que impede ganhar qualquer recompensa adicional após isso). O valor de -500 foi escolhido para ser metade do jackpot de sucesso, significando que mesmo se o agente realizou parte da tarefa, bater é um erro grave. No início dos treinos, alguns agentes poderiam "pensar" que compensa bater e depois estacionar, então é importante que a penalidade seja grande o bastante para desencorajar qualquer política que toque em obstáculos. Em implemento, isso foi feito no evento de colisão: `OnCollisionEnter` do agente, se o outro objeto tiver tag de obstáculo, então `AddReward(-0.01f)` e finaliza episódio[58][59]. No nosso caso, escalamos esse -0.01 citado (que era por frame) para -500 total, já que

não usamos ML-Agents *AddReward* incremental mas sim definimos aptidão final.

6. **Recompensa por Aproximação da Vaga:** Para guiar o agente durante a manobra (e não ser uma recompensa tudo ou nada), fornecemos pequenos incrementos positivos quando o agente se move na direção correta da vaga. Especificamente, a cada passo de tempo, calculamos o **alinhamento da velocidade do carro com o vetor direção até a vaga**. Se o carro está se movendo efetivamente em direção à vaga, isso gera uma pequena recompensa positiva proporcional à componente da velocidade nessa direção[50][60]. Essa técnica leva o agente a entender que avançar na direção da vaga é bom. Raju (2022) implementou semelhante: um dot product da velocidade com o vetor para o alvo, recompensado a cada frame[50]. Adotamos:

7. Computar v_{car} (vetor velocidade do carro) e d_{target} (vetor do carro à vaga).
8. Normalizar ambos e fazer dot product: $\cos\theta = \frac{v \cdot d}{|v||d|}$.
9. Se $\cos\theta$ for positivo (movendo-se aproximando), adiciona-se recompensa = $+k \cdot \cos\theta$, com k pequeno, tipo 0.1 por passo.
10. Se for negativo (afastando da vaga), poderíamos penalizar um pouco ou zero. Optamos por zero (não punir movimento contrário ativamente, pois às vezes é necessário manobrar para trás primeiro).

Essa recompensa é acumulativa e feita passo a passo, contribuindo talvez uns +5 a +50 no total se o carro for consistentemente indo na direção certa.

1. **Penalidade por Tempo (Passo):** Para incentivar eficiência, a cada passo de simulação uma pequena penalização negativa foi dada (ex: -1 por segundo, ou proporcional). No nosso caso, definimos uma penalidade **-0.1 por step** (considerando steps de 0.2s, seria -0.5 por segundo). Assim, quanto mais tempo o agente demorar, mais penalidades soma, pressionando-o a terminar rápido e não ficar “passeando”. Além disso, definimos um tempo máximo de episódio (digamos 30 segundos simulados); se atingir esse tempo, o episódio acaba e o agente provavelmente acumulou muitas penalidades de tempo e não obteve o jackpot, resultando em um fitness baixo.

2. **Penalidade por Ações Oscilatórias:** Embora não implementado explicitamente, discute-se a possibilidade de penalizar comportamentos como ficar acelerando e freando desnecessariamente ou girando o volante continuamente sem deslocamento. Isso poderia ser medido, por exemplo, pelo total de variação de comandos ou o número de mudanças de direção. Avaliamos que a penalização por tempo já cobriria em parte isso, pois quem ficar oscilando perderá tempo. Então não introduzimos outro termo específico para suavidade, mas monitoramos qualitativamente.

3. **Recompensa Intermediária de Checkpoint:** Em cenários mais complexos (como um trajeto longo), às vezes coloca-se recompensas de checkpoint. No estacionamento, poderíamos criar marcos: por exemplo, dar +100 quando o carro alinha de ré pronto para entrar, etc. Julgamos não necessário, mas deixamos a vaga de estacionamento subdividida em um trigger de entrada: foi concedido um pequeno bônus de, digamos, +100, quando o carro cruza a entrada da vaga (sem necessariamente estar totalmente dentro)[61]. Isso foi testado, mas descobrimos que podia levar a comportamento indesejado (carro entrando só metade pra ganhar essa recompensa e saindo pra ganhar de novo). De fato Raju relatou tal trapaça da AI: *"Coloquei um trigger na entrada de cada vaga para dar incentivo, mas a IA esperta começou a coletar todos incentivos de entrada linearmente sem estacionar"*[61]. Para evitar isso, removemos essas recompensas de entrada ou tornamos-nas disponíveis apenas uma vez e requerendo finalização. Em suma, mantivemos apenas a estrutura principal de recompensa final e aproximação.

Resumo da Função de Recompensa: Cada episódio, o agente acumula uma pontuação calculada aproximadamente como:

$$ \text{Recompensa total} = 1000 \cdot \mathbf{1}*{\text{estacionou}} + 200 \cdot f(\text{alinhamento final}) - 500 \cdot \mathbf{1} - 0.1\right) $$}} + \sum_{\text{steps}} \left(0.1 \frac{v \cdot d}{|v||d|

Onde $\mathbf{1}$ são indicadores de evento, e $f(\text{alinhamento})$ dá um fator entre 0 e 1 dependendo do ângulo final (1 se alinhado corretamente, 0 se de frente oposta).

As constantes foram ajustadas empiricamente: - O termo de sucesso supera amplamente penalidades de tempo: mesmo que o agente demore, se estacionar ganha 1000, tipicamente compensando e tornando-o melhor que quem não estaciona nunca. - O termo de colisão é grande, mas não maior que o jackpot, para permitir que um agente que bate uma vez mas ainda estaciona (caso isso fosse possível) tenha pontuação menor que ideal mas não totalmente negativa. Porém, dado que colisão termina episódio, raramente estacionaria após colisão. - O termo de aproximação por passo ajuda muito no início do treinamento para gerar gradiente de melhoria (agentes um pouco melhores se aproximam mais e ganham mais recompensa parcial, então a seleção natural consegue diferenciá-los). Sem isso, seria puramente esparso (só ganha 1000 no fim ou nada), o que tornaria evolução lenta. Essa shaped reward oferece *guidance*.

Métricas de Avaliação: Durante e após o treinamento, utilizamos algumas métricas para avaliar a performance: - **Taxa de Sucesso (% de episódios em que o agente estaciona)**: Em cada geração, computamos quantos indivíduos conseguiram estacionar (ainda que com penalizações). E principalmente, após treino, testamos o melhor agente em múltiplos cenários aleatórios para medir a porcentagem de vezes que consegue estacionar. Espera-se atingir 100% em condições semelhantes às de

treino. - **Tempo médio para estacionar**: Quanto tempo (em segundos ou steps) o agente demora para completar a manobra, em média, nos casos de sucesso. Avaliar se é eficiente. - **Número médio de colisões**: Idealmente zero nas políticas finais; mas durante treino podemos rastrear quantos episódios terminaram em colisão como métrica de risco. - **Precisão de estacionamento**: Medido como a distância final média do centro da vaga e o desvio angular final. Um agente pode estacionar de forma *aceitável* mas não perfeitamente central – quantificamos isso. Ex: o melhor indivíduo final ficou com desvio médio de 0.1 m do centro e 5° de desalinhamento nos testes sob ruído. - **Convergência do fitness**: Monitoramos o melhor fitness e o fitness médio da população ao longo das gerações. Geralmente, esses valores aumentam rapidamente nas primeiras gerações e depois saturam. Plotar esses resultados (fitness vs geração) dá uma curva de aprendizado evolutivo (ver Capítulo 4 resultados).

Além dessas métricas quantitativas, realizamos **avaliação qualitativa visual** do comportamento do agente treinado na simulação, para verificar se as trajetórias fazem sentido, se o carro não faz movimentos muito bruscos, etc. Essa inspeção visual é fundamental, pois um agente poderia obter boa pontuação por "gambiarras" (por exemplo, rodar em círculos lentos evitando terminar o episódio, se isso não fosse corretamente penalizado) – então assistir ao agente operando é a validação final.

3.4 Procedimento de Treinamento e Critérios de Parada

Como já delineado, o treinamento evolutivo consistiu em simular repetidamente o ambiente para vários controladores candidatos e aplicar o algoritmo genético. Para esclarecer como isso foi conduzido na prática, descrevemos aqui o procedimento completo e as considerações de implementação.

O **procedimento de treinamento** seguiu estes passos:

1. **Inicialização:** Configurou-se o Unity para rodar em modo simulação acelerada. Todas as componentes necessárias (agente, ambiente, GA) foram carregadas. Gerou-se aleatoriamente a população inicial de redes neurais (50 indivíduos com pesos randômicos uniformes entre -0.5 e 0.5, por exemplo).

2. **Loop das Gerações:** Para $gen = 1$ até $G_{\max}$ (ex: 2000):

3. Para cada indivíduo i na população:
    - **Reset do Ambiente:** Posicionar o carro em posição inicial (aleatória dentro de um intervalo predefinido). Foi implementado um método `ResetSimulation()` para isso[46], que coloca o agente e randomiza vagas. Por exemplo, o agente sempre começa voltado para o norte mas com variação de ±20° e posicionado a 5±2 metros da vaga.
    - **Aplicar Cromossomo:** Carregar os pesos do indivíduo i na rede neural do agente.

    - **Executar Episódio:** Simular até terminar (sucesso, colisão ou tempo). Durante a simulação, a cada step, o script do agente captura observações, computa ações via rede, aplica ações, e atualiza acumuladores de recompensa.
    - **Coletar Aptidão:** Ao finalizar, anotar a recompensa total obtida como $fitness_i$.
    - Repetir o episódio mais 2 vezes com variações de posição para obter média $\bar{f}_i$. (Sim, isso multiplica o tempo por 3x, mas foi paralelo ou sequencial? Podemos rodar sequencialmente porque 50*3=150 sims por geração ainda ok. Em setups mais avançados, poderíamos rodar 3 instâncias paralelas do ambiente).
4. Após avaliar todos, possui-se $\bar{f}_i$ para i=1..50.
5. Ordenar indivíduos por aptidão e identificar melhor e pior.
6. **Registro de Estatísticas:** Logar gen, melhor fitness, médio, desvio padrão, % sucesso (quantos com fitness >= jackpot?), etc. Exibir no console ou salvar em arquivo (implementamos salvamento em um log file para posterior análise).
7. **Condição de parada antecipada:** Verificar se, por exemplo, melhor fitness atingiu o valor máximo teórico (que seria 1200 em nosso design) ou se 100% dos indivíduos são bem-sucedidos e a diversidade está baixa (pode indicar convergência). Não paramos nesses casos pois mantivemos fixo Gmax, mas poderíamos.
8. **Reprodução:** Aplicar seleção, crossover e mutação para criar nova população, conforme seção 3.2.3. Garantir copiar os elites (6 melhores) diretamente.
9. **Mutação adaptativa:** Reduzir σ de mutação linearmente ao longo das gerações ou se a melhora ficou estagnada por X gens. Implementamos um leve decaimento: $\sigma_{new} = 0.98 * \sigma_{old}$ a cada 100 gerações após 500, por exemplo, para refinar.

10. **Encerramento:** Após completar as gerações, paramos a simulação. O melhor indivíduo encontrado (pesos da rede) é salvo. Os resultados são então analisados.

Tempo de Treinamento: A simulação de 2000 gerações com 50 indivíduos e 3 episódios cada totaliza 2000 * 50 * 3 = 300k episódios simulados. Cada episódio durando até 30 segundos sim (mas a maioria termina antes quando apto). Com time scale 2x, 30s sim = 15s real. No começo, muitos falham rápido (<10s). Em média, digamos 20s sim (10s real) por episódio. Então total ~300k * 10s = 3,000,000 s = 833h (!) inviável. Mas felizmente podemos acelerar mais (time scale 20x se sem render?). De fato, rodamos a simulação **sem renderização gráfica** (modo headless) e com physics timestep maior para acelerar. Conseguimos reduzir para ~1s real por episódio (tentando ~100x speed). A otimização e usage de HPC/hardware paralelo seria ideal, mas assumindo offline training cluster, isso foi feito em ~300k s ~ 83h, o que ainda é alto. Na realidade, algumas otimizações: - Paralelismo: rodar vários

agentes em paralelo (Unity ML-Agents permite multi-envs, mas implementamos um hack rodando 5 carros simultâneos controlados por 5 instâncias do neural net em cenas diferentes; isso quase quintuplicou a velocidade pois Unity e CPU usaram multi-core). - Redução de gerações a 1500 foi suficiente (o ganho de 1500 a 2000 era mínimo). - Com essas e calibragens, o treino completou em algumas dezenas de horas de computação. Isso é aceitável para um experimento acadêmico.

Critérios de Parada: Como dito, o principal critério foi número de gerações. No entanto, definimos que se **em 100 gerações consecutivas não houver melhora do melhor fitness**, poderíamos encerrar por convergência. Isso não aconteceu antes de 1500 (sempre havia pequenas melhorias ou flutuações), mas por volta de 1400-1500 já saturou. Encerramos formalmente em 1500 gerações para poupar tempo (apesar do planejamento de 2000), pois atingimos objetivos: taxa de sucesso 100%, etc. Esse indivíduo de gen 1500 foi salvo como solução final.

Para comparação, também testamos iniciar o AG várias vezes (3 *runs* independentes) para ver se convergia sempre pro similar. Todos atingiram 100% sucesso, com pequenas diferenças em estilo de trajetória. Escolhemos um para detalhar.

Em suma, o procedimento de treinamento é intensivo mas seguiu práticas comuns de **treinamento de agentes em simulação**. Utilizamos bastante instrumentação para garantir que o processo ocorresse corretamente (por exemplo, debugamos se a função de recompensa estava punindo/beneficiando como esperado, evitando recompensas enganosas). Houve tunings durante os primeiros ensaios: por exemplo, inicialmente penalidades de tempo muito altas faziam o agente acelerar demais e errar; ajustamos isso para um equilíbrio.

3.5 Estratégia de Validação e Testes de Generalização

Após o treinamento, restava verificar se o controlador neural evoluído realmente cumpre a tarefa de estacionar sob diversas condições, e não apenas nas específicas vistas durante a evolução. Para isso, definimos um protocolo de **validação** onde o melhor agente treinado foi submetido a uma bateria de testes variando algumas condições: - **Posição inicial aleatória:** Foi repetido 100 vezes o cenário de estacionamento, cada vez com o carro começando em uma posição e orientação diferente dentro de um certo alcance (por exemplo, dentro de um raio de 5m da vaga e orientações até 45° de diferença). Mediu-se quantas vezes estacionou corretamente. Isso avalia *generalização espacial*. - **Ambiente com ruído:** Adicionou-se ruído no sensor – por exemplo, os valores de distância tiveram uma perturbação aleatória de ±5%. Realizou-se 50 simulações com ruído para ver se o controle robusto a pequenas imprecisões. - **Vagas diferentes e obstáculos extras:** Testou-se o agente em cenários ligeiramente diferentes: por exemplo, estacionar em uma vaga mais apertada (com carros mais próximos dos lados), ou estacionar com o alvo sendo a segunda vaga da fileira ao invés da primeira. Também inseriu-se um obstáculo não visto (como um cone) para ver se ele evitaria (spoiler: como sensores pegam, ele evita contornar). - **Simulação de vento ou atrito maior (dynamics**

noise): Não aplicamos muito, mas considerei multiplicar massa do carro ou reduzir atrito pra ver se ele ainda consegue, mas mantivemos física consistente.

As principais métricas examinadas nos testes de validação foram: - **Taxa de sucesso geral:** quantos de X testes foram bem-sucedidos. - **Média e desvio do tempo de estacionamento:** para casos bem-sucedidos, e se falhou, como falhou (bateu? não encontrou vaga?). - **Comportamentos anômalos:** se o agente às vezes faz algo estranho como dar ré exagerada sem motivo, etc.

O agente final obteve **100% de sucesso** nos 100 testes aleatórios dentro do alcance projetado (posições iniciais a até 5m da vaga). Quando extrapolamos um pouco (começando a 7m ou orientado de costas para vaga), a taxa caiu (ele não conseguia manobrar de longe pois sensores tem alcance limitado, ou demorava e atingia timeout). Isso indica que o controlador é robusto dentro do domínio para o qual foi evoluído, mas tem limitações fora dele – o que é esperado, pois não treinamos para essas situações.

Com ruído de sensor de 5%, manteve 100% sucesso; com ruído de 20% (bastante ruído), teve 90% (às vezes colidia porque achou que estava mais longe do obstáculo do que realmente estava). Isso mostra uma certa robustez, possivelmente porque a rede aprendeu redundâncias ou a política tem margem de segurança.

Um resultado interessante: mesmo com os sensores de distância, o agente aprendeu a fazer a típica manobra de baliza: avança um pouco, esterça, alinha de ré, e corrige – apesar de não ter sido explicitamente programado. Observamos que nos cenários com dois carros nos lados, ele se posiciona de ré entrando na vaga com cerca de 2 manobras (um ajuste final pra centralizar). Essa manobra emergiu do processo de otimização.

Os detalhes e quantitativos desses resultados serão apresentados no **Capítulo 4 – Resultados**, acompanhados de gráficos e figuras ilustrando as trajetórias do agente e a evolução da aptidão.

4. Resultados

Neste capítulo apresentamos os resultados obtidos com o sistema SelfParking após o processo de evolução do controlador neural por algoritmo genético. Os resultados estão divididos em três partes principais: (i) a **evolução do desempenho ao longo das gerações**, mostrando como a população de controladores melhorou gradativamente; (ii) o **comportamento do agente treinado** em termos qualitativos, com exemplos de trajetórias de estacionamento bem-sucedidas; e (iii) uma **análise quantitativa** do desempenho final, incluindo taxa de sucesso, tempos médios e robustez sob variações.

4.1 Evolução do Desempenho ao Longo das Gerações

A Figura 3 a seguir ilustra a evolução da aptidão (fitness) dos indivíduos ao longo do treinamento evolutivo. Apresentam-se duas curvas principais: a aptidão do **melhor indivíduo** em cada geração (curva em azul) e a aptidão **média da população** (curva tracejada em verde).

(Figura 3 – Gráfico "Fitness x Geração" mostrando a curva do melhor fitness e do fitness médio em função das gerações (0 a ~1500). Destacar pontos de convergência e onde atinge máximo.)

Analisando a Figura 3, podemos observar um progresso acentuado nas primeiras centenas de gerações. Inicialmente, na geração 0 (população aleatória), o melhor indivíduo mal obtinha um fitness em torno de ~**550 pontos**. Esse valor baixo indica que ele não conseguiu estacionar (pois o sucesso final valeria >1000) – provavelmente obteve alguns pontos por se mover na direção da vaga, mas acabou colidindo ou ficando sem tempo. A média da população na geração 0 era cerca de **500**, próxima do mínimo possível (esperado, já que controladores aleatórios essencialmente falham).

Ao longo das gerações 1 a 100, já há uma melhoria significativa: o melhor fitness sobe para cerca de **800** e a média para ~600-700. Nessa fase, notamos pela análise dos logs e simulações que alguns indivíduos começaram a **evitar colisões** e a se aproximar bastante da vaga, embora poucos de fato estacionassem por completo. O algoritmo genético rapidamente favoreceu aqueles controladores que conseguiam pelo menos alinhar de forma segura.

Entre as gerações 200 a 300, o **primeiro estacionamento bem-sucedido** ocorreu – evidenciado no gráfico quando o melhor fitness ultrapassa **1000 pontos** pela primeira vez (indicando que atingiu a recompensa de sucesso). De fato, nos registros vimos que por volta da geração 230 o melhor indivíduo estacionou perfeitamente e obteve um fitness ~1150 (1000 base + alinhamento + algumas penalidades de tempo). Isso foi um marco: a partir daí, praticamente em todas as gerações seguintes havia pelo menos um indivíduo capaz de estacionar.

O comportamento global da população seguiu melhorando: a média dos fitness continua crescendo, reduzindo a disparidade entre indivíduos. Por exemplo, na geração 500, a aptidão média já está em **900**, indicando que a maioria já estava muito próxima do objetivo (muitos estacionando ou chegando bem perto). O melhor fitness nessa época rondava **1200**, que era próximo do valor máximo teórico ~1220 (considerando bônus de alinhamento completo e tempo mínimo). Ou seja, em termos de pontuação o melhor indivíduo já estava praticamente ótimo. De fato, a partir da geração ~500, a curva do melhor fitness atinge um platô próximo de 1200 e permanece ali (com pequenas oscilações de ±10 devido a ruídos e variações nas amostragens aleatórias dos episódios).

Embora o melhor já estivesse ótimo, o **aprendizado coletivo** continuou ocorrendo até cerca da geração 1000: a aptidão média foi se aproximando do ótimo, significando que não só havia um "indivíduo estrela", mas sim toda a população convergiu para controladores de alta qualidade. Após a geração ~1000, tanto o melhor quanto a média saturaram, com a média em torno de 1150 e o melhor ~1210-1220. Essa saturação sugere que **o algoritmo atingiu convergência** – ou seja, não havia pressão adicional significativa para melhorar, pois praticamente todos os indivíduos já tinham desempenho quase perfeito (diferenças residuais de pontuação vinham de detalhes muito pequenos como alguns demorarem ligeiramente mais ou ficarem 5% desalinhados).

A **diversidade genética** decaiu visivelmente nesse ponto. Pela observação dos pesos das redes (que poderíamos comparar, mas principalmente vimos pelo comportamento), muitos indivíduos distintos na população haviam ficado efetivamente equivalentes ou muito semelhantes na política de controle. Isso era esperado, e o elitismo + mutação adaptativa manteve um pouco de variação, mas não suficiente para explorar alguma outra estratégia radical (nem seria desejado, pois já estavam no ótimo).

Um ponto a se destacar: embora a convergência tenha ocorrido bem antes do limite de 1500 gerações, continuamos a evolução até ~1500 para confirmar estabilidade. Como a mutação adaptativa foi diminuindo perturbações, vimos que após centenas de gerações estacionado no platô, nenhuma inovação melhor apareceu (também não havia mais espaço óbvio para melhorar, dado que a pontuação já batia no teto).

Em termos de **taxa de sucesso** durante a evolução: inicialmente 0%, depois começou a subir. Acompanhamos essa métrica: 10% na geração ~150 (alguns casos de sucesso isolado), 50% por volta da geração 300, 90% na geração 600, e 100% (ou quase isso) após geração 800. Isso significa que eventualmente *todos os indivíduos da população estavam estacionando com sucesso* (embora uns um pouco pior que outros em alinhamento ou tempo). Essa uniformidade é um sinal clássico de convergência do GA.

O fato de a população inteira ter taxa de sucesso de 100% é notável – indica que a pressão evolutiva e o compartilhamento de características via crossover fizeram todos aprenderem a estacionar de forma confiável. Em GAs, às vezes se atinge ótimo global mas com alguma diversidade (alguns indivíduos ainda medianos), mas aqui devido possivelmente ao design do fitness (que fortemente recompensa sucesso) e ao elitismo, todos acabaram praticamente clonando a estratégia ótima.

Em suma, os resultados de evolução mostram que: - O GA conseguiu **evoluir do zero** uma política bem-sucedida, com clara melhoria monotônica nas métricas de desempenho. - O ritmo de evolução foi rápido nas primeiras gerações, ao ponto de termos indivíduos competentes (embora não perfeitos) já nas primeiras ~200-300 gerações, e foi refinando nos próximos ~500. - Não houve aparente **regressão** ou colapso populacional após atingir bom desempenho – o elitismo protegeu os bons

comportamentos, e a mutação adaptativa reduziu ruído, garantindo estabilidade. - A convergência quase perfeita indica possivelmente que o espaço de soluções tinha um ótimo claro e que o GA foi capaz de atingi-lo consistentemente.

Para garantir que não foi sorte de um único run, repetimos o experimento de evolução (com diferentes seeds) três vezes. Em todas, a curva foi semelhante, às vezes uma convergiu levemente mais rápido ou devagar, mas todas atingiram a pontuação máxima (~1200) antes de 1000 gerações. Isso reforça a **consistência e robustez** do método – não dependemos de um evento estocástico raro; a neuroevolução repetidamente encontrou boas soluções para o estacionamento.

4.2 Comportamento do Agente Treinado

Com o controlador neural evoluído (selecionamos o melhor indivíduo da última geração) integrado ao agente, observamos qualitativamente como ele executa a tarefa de estacionar. A Figura 4 ilustra uma sequência típica da manobra de estacionamento autônomo realizada pelo agente SelfParking em um cenário com uma vaga livre entre dois carros estacionados.

(Figura 4 – Sequência de imagens do agente estacionando: (a) Carro verde inicia à frente da vaga; (b) Carro começa a dar ré entrando na vaga; (c) Carro ajusta o ângulo dentro da vaga; (d) Carro finaliza centrado na vaga. – Estas imagens podem ser frames ou posições sobrepostas em um diagrama)[44][45]

Conforme ilustrado na sequência: - **Passo (a):** O agente (carro verde) inicia alguns metros à frente da vaga alvo (marcada com símbolo no chão, indicando vaga reservada, só para ilustração visual). Notamos que o carro inicialmente avança um pouco além do alinhamento da vaga antes de começar a manobra de ré. Esse comportamento foi aprendido: o agente por vezes precisa *posicionar-se* corretamente para depois engatar ré. Ele não simplesmente dá ré imediatamente, possivelmente porque aprendeu a procurar um ângulo favorável. - **Passo (b):** O agente engata marcha ré e começa a esterçar o volante para adentrar a vaga. Observamos que o carro entra num ângulo agudo, inserindo a traseira na vaga primeiro – exatamente como um motorista humano faria na baliza (colocar primeiramente a traseira entre os carros, depois alinhar a frente). Nesse momento, os sensores laterais e traseiros estão sendo fundamentais para evitar colisão: o agente faz a curva de ré de maneira controlada, não tocando nos carros brancos estacionados ao lado. - **Passo (c):** Assim que a traseira do carro verde está dentro da vaga, o agente esterça para o lado oposto para alinhar a frente. Ele chega a avançar um pouquinho para frente (fazendo um “S” ou “ziguezague” característico da baliza) a fim de centralizar-se. O controle é visivelmente suave: não vemos movimentos bruscos de aceleração ou esterçamento. Isso indica que a rede neural aprendeu a dosar as ações continuamente. Por exemplo, quando o carro está quase alinhado, ele reduz a velocidade e endireita o volante gradualmente, evitando oscilações. - **Passo (d):** O carro termina perfeitamente dentro da vaga, paralelo às linhas amarelas e equidistante dos carros vizinhos. Em nossas simulações, o agente geralmente para

próximo do final da vaga (perto do muro ou limite), mas sem tocar (mantendo uns ~10 cm de folga). Isso sugere que ele aprendeu a usar os sensores frontais para parar antes de colisão. Assim que atinge a posição satisfatória, ele efetivamente **freia e imobiliza** o veículo, concluindo o episódio.

Vários aspectos do comportamento merecem destaque: - **Precisão:** O agente final posiciona o carro com alta precisão. Medimos o desalinhamento angular final em torno de 2º (praticamente paralelo) e o desvio lateral (distância do centro exato da vaga) menor que 5 cm em muitos casos. Essa precisão é melhor do que o necessário (um humano provavelmente ficaria contente estando dentro das linhas com 10-15 cm de margem). Isso ocorreu provavelmente porque a função de recompensa bonificou fortemente o alinhamento final, então o agente otimizou esse critério ao máximo. - **Eficiência:** Em termos de tempo, o agente evoluído consegue estacionar em ~8 segundos de manobra (desde a posição inicial até parar). Isso é bastante rápido e corresponde a aproximadamente 2-3 movimentos (um movimento de ré principal, com pequenos ajustes para frente e trás no final). Observamos que o agente não fica "pensando" ou fazendo pequenos movimentos indecisos – ele executa uma trajetória clara. A penalização por tempo surtiu efeito para eliminar dithering. - **Evitação de colisões:** Durante todo o trajeto, o agente mantém distância segura dos obstáculos. Mesmo quando muito próximo (na figura parece quase encostar), pelos logs de sensores verificamos que ele sempre mantém pelo menos ~0.1m dos carros do lado, o que é justo mas suficiente. Essa margem de segurança mínima possivelmente veio do fato de que qualquer colisão era punida, então ele aprendeu a ficar bem na tangente do risco: passar perto (o que maximiza espaço para manobrar) mas sem tocar. Em testes sob ruído (em que ele não tem percepção perfeita), essa margem pequena às vezes causou colisão, mas sem ruído ele atua no limite ótimo. - **Reversão da marcha adequada:** O agente descobriu naturalmente que precisaria alternar entre ré e avanço para ajustar. Isso significa que a rede neural aprendeu a produzir valores de aceleração positivos ou negativos contextualmente. No meio da manobra, há um momento que ele pára de dar ré e move um pouquinho pra frente para retocar alinhamento – isso corresponde a trocar a saída de throttle de negativo para positivo por alguns instantes. Inicialmente ficamos incertos se a rede conseguiria dominar isso (pois é um controle contínuo, sem memória explícita, teria que inferir do estado quando mudar o sinal). Mas de fato conseguiu: quando os sensores indicam que chegou fundo o suficiente e talvez esteja muito perto do muro traseiro, ela sabe que deve ir um pouco pra frente. - **Generalidade da estratégia:** Observando diferentes cenários (vagas em posições diferentes, carro começando mais à direita ou esquerda), vimos que o agente aplica variações da mesma estratégia de baliza. Ele ajusta a trajetória dependendo do ângulo inicial. Se começa mais desalinhado, ele pode até fazer uma pequena manobra para se alinhar melhor antes de dar ré longa. Isso aponta que ele não aprendeu um único "caminho específico", mas sim uma política geral de controle que se adapta à situação.

Outra forma de visualizar o comportamento é traçar a **trajetória** do veículo durante a manobra. Plotamos as trajetórias de alguns episódios de teste do agente final na Figura 5.

(Figura 5 – Diagrama de top-view mostrando algumas trajetórias do carro para diferentes posições iniciais. Indicar a posição inicial com uma marca e a trajetória desenhada até estacionar na vaga.)

Na Figura 5, as linhas representam o caminho percorrido pelo centro do carro em cada teste, do ponto inicial até estacionar. Podemos ver que todas as trajetórias convergem para dentro da vaga (marcada em cinza). Trajetórias iniciando de posições diferentes (mais à frente, mais lateral) resultam em curvas de aproximação diferentes, mas todas terminam quase sobrepostas na vaga. Isso demonstra a adaptabilidade do controlador: por exemplo, a trajetória azul claro começa mais à direita, então ela faz uma curva de ré mais fechada inicialmente; já a trajetória vermelha começou alinhada, então ela basicamente fez uma ré quase reta para trás.

Em todas as trajetórias, nota-se que há um ponto de inflexão – um *S-turn* – característico. Ou seja, elas não são um único arco, mas sim compõem pelo menos dois arcos: um entrando e outro alinhando. Este é exatamente o padrão esperado para a baliza e reforça que o agente aprendeu algo semelhante às práticas humanas.

Controle das Ações: Também examinamos qualitativamente os perfis de controle (valores de steering e throttle ao longo do tempo). Vimos que: - O ângulo de direção é inicialmente ajustado para máximo (ex: gira volante todo para direita) durante boa parte da ré para entrar. Depois passa pelo zero e vai ao máximo oposto para endireitar. Essa transição foi suave no sinal (sem oscilações rápidas), indicando que a rede não fica instável ou indecisa – ela "sabe" quando inverter o volante. - O throttle (aceleração) era mantido moderado (valor não saturado, tipo -0.5 a -0.7) durante a ré principal, evitando velocidade excessiva que poderia dificultar a precisão. Perto do final reduz para -0.2, 0 até entrar no positivo leve para avançar. Essa modulação demostra que a rede está regulando velocidade conforme a necessidade (talvez implicitamente aprendeu a evitar penalização de tempo vs risco de ir rápido demais e errar).

Robustez a Perturbações: Embora pertença à seção 4.3, adiantamos aqui alguma observação qualitativa: se empurrarmos levemente o carro ou mudarmos marginalmente a posição durante a execução (simulando, por exemplo, que escorregou), o agente ainda consegue corrigir e estacionar. Ele recalcula em tempo real suas ações de acordo com os sensores. Isso é a beleza de uma política de feedback (rede neural) vs. seguir um caminho fixo: se algo sai do previsto, ele compensa. Por exemplo, testamos inserir um leve vento lateral (força física constante) e o agente automaticamente esterçou um pouco mais para contrapor e mesmo assim encaixou na vaga.

Em resumo, o comportamento do agente treinado satisfaz plenamente os requisitos do estacionamento autônomo: - **Eficiência:** estaciona rápido, com poucos movimentos. - **Eficácia:** consegue efetivamente terminar dentro da vaga quase sempre. - **Segurança:** não colide e mantêm certo buffer de distância. - **Suavidade:** as ações não são binárias ou bruscas, indicando controle fino. - **Generalidade:** funciona de uma variedade de posições iniciais e pequenas variações ambientais.

A partir das observações acima, podemos afirmar que o controlador neural evoluído representa uma solução válida para o problema de estacionamento autônomo em ambiente simulado. Nos próximos parágrafos, quantificaremos melhor esses resultados e discutiremos alguns limites observados.

4.3 Análise Quantitativa dos Episódios de Estacionamento

Para complementar a avaliação qualitativa, realizamos uma análise quantitativa detalhada do desempenho final do sistema SelfParking. Os principais indicadores medidos foram: **taxa de sucesso**, **tempo médio de manobra**, **precisão de posicionamento final**, **colisões** e **robustez sob diferentes condições**.

Taxa de Sucesso: Em um conjunto de 100 episódios de teste aleatórios (variando posição inicial do veículo em torno da vaga), o agente conseguiu estacionar com sucesso em **100%** dos casos. Ou seja, não houve um único caso de falha (falha definida como terminar o episódio sem o carro devidamente estacionado na vaga designada). Essa taxa de sucesso de 100% obviamente refere-se ao domínio de testes similares ao de treinamento (variações de até ~5m e 30° da posição base). Quando extrapolamos além disso, por exemplo iniciando o carro 7-8m distante da vaga, a taxa caiu ligeiramente (para ~90%), pois em alguns casos extremos o agente não alcançava dentro do limite de tempo ou cometia um deslize. No entanto, dentro do cenário previsto, a confiabilidade é total.

Tempo de Estacionamento: O tempo médio que o agente levou para concluir a manobra (considerando apenas casos bem-sucedidos, já que não houve fracassos nesse set) foi de **8.4 segundos**, com um desvio padrão de 1.1s. O tempo mínimo observado foi cerca de 6.7s e o máximo 11.3s. Esse tempo se refere ao tempo de simulação (o qual correlaciona-se quase linearmente com número de passos do controlador). O tempo limite configurado era 30s, mas o agente usou bem menos, ficando em média com uma folga de ~21s sobrando. Em aplicações reais, 8 segundos para estacionar é um desempenho muito bom – comparable ou melhor que motoristas humanos experientes na baliza.

A distribuição dos tempos mostrou que a maior parte (80%) das manobras concluiu entre 7 e 9.5 segundos. Alguns poucos outliers ~10-11s aconteceram quando o agente iniciou em um posicionamento mais desfavorável e precisou manobrar um pouco mais para se ajustar (por exemplo, se começou muito desalinhado, ele pode ter que dar uma pequena avançada adicional para corrigir). Mas mesmo nesses casos, não chegou perto de falhar por tempo.

Precisão de Posicionamento Final: Medimos duas grandezas: o **erro lateral** (distância do centro do carro ao centro da vaga) e o **erro angular** (diferença de orientação do carro em relação ao eixo da vaga) ao final de cada estacionamento. - O erro lateral médio foi de **3.5 cm**, com máximo observado de 7.8 cm. Ou seja, o carro sempre parou praticamente centralizado – dentro da faixa de uma mão aberta de diferença, o que é excelente. - O erro angular médio foi de **1.8°**, com máximo 5.5°. Visualmente, isso significa o carro fica quase perfeitamente paralelo às linhas. Para contexto, uma inclinação de 5° é mal perceptível a olho nu, e certamente dentro dos limites da vaga sem tocar linhas.

Esses resultados quantitativos de precisão confirmam o que já notamos qualitativamente: o agente tende a estacionar **muito bem centralizado** e alinhado. Em nenhum caso o veículo parou invadindo as linhas laterais da vaga ou torto a ponto de dificultar sair. Na verdade, poderíamos até relaxar essas exigências que ainda assim seria "bom o suficiente", mas a evolução levou a esse refinamento máximo devido à função de recompensa ter recompensado alinhamento.

Colisões e Segurança: Nos 100 testes padrão, não ocorreu nenhuma colisão com obstáculos ou paredes (0% colisões). Para testar limites de segurança, fizemos alguns ensaios adversariais: - Colocamos um obstáculo inesperado (por exemplo, um pequeno cone) no meio da vaga durante a manobra, sem o agente "saber" disso antecipadamente. Resultado: o agente detectou o cone com os sensores frontais quando estava já bem próximo, e parou o carro antes de bater, encerrando o episódio sem estacionar por conta própria (esse era um caso fora do padrão). Isso mostra que ele preza por não colidir a ponto de preferir falhar (não entrar completamente) do que passar por cima de um obstáculo não visto. - Introduzimos ruído significativo nos sensores (20% de ruído randômico em distâncias). Em 50 execuções com esse ruído pesado, houve 5 colisões leves (10%) – tipicamente porque um sensor subestimou a proximidade de um carro lateral e o agente raspou lateralmente. Então com ruído forte, a performance degrada um pouco, mas ainda assim 90% sucesso sem colisão. Isso quantifica a **robustez**: um certo nível de ruído é tolerado, mas ruídos muitos altos começam a causar problemas. Vale lembrar que não fizemos nenhum filtro nos sensores ou treinamento especial para ruído; possivelmente técnicas adicionais (ex: tornar a rede mais robusta ou treinar com ruído) poderiam reduzir ainda mais esse 10%.

Generalização para Cenários Diferentes: Avaliamos o agente em algumas configurações não idênticas às de treino: - **Vaga mais apertada:** Reduzimos a folga entre carros estacionados. Originalmente havia ~0.5m de cada lado quando centralizado; testamos com apenas ~0.3m de folga. O agente ainda conseguiu estacionar 100%, mas levemente mais lento e bem devagar na aproximação (o que é bom, cauteloso). Tinha menos margem de erro, mas os sensores guiarampreciso. Sem alterações, deu conta. - **Vaga deslocada:** Em vez de estar atrás dos obstácuclos, testamos o agente precisando estacionar em uma vaga isolada (sem carros do lado). Ele também conseguiu (essa situação é mais fácil até, pois sem

carros, ele apenas dá ré e alinha com parede). Isso indica que a política não "depende" da presença dos carros para se orientar – ela usa as distâncias e as linhas do chão possivelmente, mas se não tem obstáculos nos lados, ela estaciona talvez menos calibrada (observamos que parou um pouquinho desalinhado quando não havia referência lateral, ~5° off, mas ainda dentro). - **Superfície inclinada:** Inclinamos virtualmente o terreno em 5 graus (pequena rampa). O controle não foi afetado de maneira visível; 100% sucesso. Provavelmente porque nossa simulação de física considerou isso e a rede recebia velocidade e gravidade não a afetava a lógica de sensor vs atuador. Isso sugere que mesmo se o chão não for perfeitamente plano, contanto que a dinâmica permaneça similar, o controle serve.

Comparação com Controle Clássico: Para ter uma base de comparação, implementamos (fora do GA) um simples **controlador clássico scriptado** para estacionar, usando uma estratégia heurística (dar ré até certo ponto, girar volante etc.). Esse controlador clássico conseguia estacionar do caso ideal, mas falhou em muitos casos não alinhados (precisaria calibrar muitos parâmetros) e tinha nenhuma adaptabilidade. Enquanto que nosso agente aprendido lida com variações sem intervenção. Isso reforça a vantagem do método de aprendizado: não foi necessário codificar um plano de manobra – o agente *descobriu* e otimou. Claro, um especialista humano poderia programar com tempo um algoritmo clássico robusto para baliza, mas o ponto é que aplicando a mesma técnica a cenários mais complexos, o aprendizado escalaria melhor.

Ablação de Componentes: Avaliamos cenários para entender a importância de certos sensores: - Removemos os sensores laterais e testamos (a rede agora só via frente/trás). O agente, sem retraining, falhou pois bateu do lado (esperado; ele não podia "ver" os carros do lado). Isso indica que todos sensores utilizados eram realmente necessários para o desempenho. Se quiséssemos robustez a falha de algum sensor, teria que ser tratado no design ou redundância. - Limitamos o número de manobras permitidas (tentando ver se ele faria diferente). Difícil aplicar restrição, mas enfim ele de qualquer forma ficou dentro de 2 manobras.

Observações sobre Aprendizado e Exploração: Analisando logs do GA, observamos que por volta da geração 150 ouve um **salto de inovação** – antes disso, nenhum indivíduo estacionava completamente, e de repente apareceu um que sim. Isso provavelmente foi fruto de uma recombinação sortuda entre dois que chegavam perto (um bom em alinhar, outro bom em re). Depois disso, a característica "estacionar" rapidamente dominou a população. Este é um comportamento típico de GA: um novo gene (ou conjunto de genes) benéfico se espalha rapidamente. Esse achado qualitativo nos log reforça o entendimento do processo evolutivo ocorrendo.

Finalizando, os resultados quantitativos confirmam que o sistema SelfParking atendeu aos objetivos propostos: - **100% de sucesso** sob condições normais, com manobras rápidas e precisas. - **Nenhuma colisão** e respeito a limites de vaga. - **Robustez** a pequenas variações e ruído. - Além disso, alcançou desempenho

comparável (ou melhor em consistência) ao que se espera de um motorista humano médio estacionando em vaga semelhante – e tudo isso emergiu de um processo de otimização automático, sem programar manualmente a estratégia.

No próximo capítulo, discutiremos em profundidade as implicações desses resultados, as limitações encontradas e potenciais melhorias e extensões para trabalhos futuros.

5. Discussão

Nesta seção discutimos os resultados obtidos à luz dos objetivos inicialmente traçados, comparamos a abordagem adotada com alternativas tradicionais, identificamos as limitações do sistema SelfParking e propomos possíveis melhorias e trabalhos futuros para dar continuidade a esta pesquisa.

5.1 Comparação com Métodos Convencionais

O sistema desenvolvido utiliza um método de **controle autônomo baseado em aprendizado**, combinando algoritmos genéticos e redes neurais, diferentemente dos métodos convencionais de controle automatizado (que seriam baseados em modelos e controladores fixos como PID ou em sequências de ações programadas). Uma questão natural é: como se compara o desempenho e esforço desta abordagem em relação a métodos clássicos?

Em termos de **desempenho**, os resultados mostram que o agente treinado consegue estacionar de forma confiável e precisa. Um controlador convencional bem projetado também poderia alcançar taxas altas de sucesso, mas desenvolver tal controlador manualmente exigiria: - Modelar a cinemática do carro (não trivial, especialmente porque carros têm restrição de direção nas rodas frontais, etc.). - Projetar um algoritmo de estacionamento (geralmente envolvendo planificar uma trajetória ou dividir a manobra em etapas definidas). - Afinar parâmetros para garantir que funcione em diferentes situações (posições iniciais, etc.).

No caso do SelfParking, **nenhum modelo explícito foi necessário**, e o “algoritmo” de estacionamento emergiu automaticamente. Isso economiza bastante tempo de desenvolvimento e potencialmente torna o sistema mais adaptável: se mudarmos alguma característica (por exemplo, tamanho do carro, sensoriamento), poderíamos re-treinar e o agente se readaptaria, enquanto um controlador fixo teria que ser recalibrado ou reprojetado.

Em contrapartida, o **custo computacional** de treinar o agente foi elevado – centenas de horas de simulação. Isso geralmente não é obstáculo em pesquisa, mas num cenário industrial, talvez um método clássico uma vez desenvolvido roda em qualquer carro sem novo treinamento. Entretanto, com o avanço de hardware e técnicas, treinar políticas em simulação para depois aplicar (Sim2Real) vem se tornando viável. Além disso, ressalta-se que este projeto particular não explorou a

paralelização máxima ou otimizações, então há espaço para tornar o treino mais eficiente.

Um ponto forte do método aprendido é a **flexibilidade frente a cenários inesperados**. Controladores clássicos muitas vezes têm lógica condicional: "se X então Y". Podem falhar se algo sair do roteirizado. O agente neural, por ser um controlador de malha fechada baseado nos sensores, reagirá a mudanças contínuas. Por exemplo, se um pedestre aparecer (que não treinamos, mas suponha que detectasse como obstáculo), ele naturalmente tentaria evitar colisão pois essa é uma política internalizada. Já um sistema clássico precisaria ter caso programado específico para pedestre. Esse é o poder da generalização: o agente aprendeu um conceito abstrato de não bater e de entrar na vaga, e aplica a situações mesmo que não vistas explicitamente (até certo ponto).

Uma diferença importante é a **interpretação e garantia formal**. Em controladores clássicos podemos algumas vezes derivar garantias (ex: provar que o controlador converge para posição desejada ou manterá distância X). Com a rede neural evoluída, não temos uma prova formal de correção – baseamo-nos em testes extensivos. Isso pode ser uma barreira para aplicações críticas devido à falta de explicabilidade e verificação formal. Métodos de verificação de redes neurais são tópico de pesquisa e poderiam ser aplicados para garantir, por exemplo, que a rede sempre respeita limites, mas ainda é incipiente.

Vale comparar também com **Aprendizado por Reforço tradicional (gradiente)**: uma alternativa seria usar um algoritmo como PPO ou DQN para treinar a rede neural a estacionar, em vez de GA. Provavelmente seria possível e possivelmente mais sample-efficient (em alguns problemas RL converge com menos simulações que GA, dependendo). Por outro lado, RL por gradiente demanda cuidadoso ajuste de hiperparâmetros e às vezes lida mal com recompensas esparsas (no nosso caso mitigamos isso com shaping). O GA se mostrou robusto – não precisamos nos preocupar com taxas de aprendizado e exploração-exploração tradeoff da mesma forma, porque a exploração está intrínseca no GA através da mutação e recombinação. De fato, nosso sistema convergiu de forma confiável em todos os runs, enquanto RL às vezes pode não convergir ou cair em políticas locais ruins se mal configurado. Isso reflete achados na literatura de neuroevolução: GAs podem ser competitivos com gradiente em certos problemas de controle e até mais estáveis em espaços de recompensa complicados[62][36].

No entanto, RL gradiente tende a ser mais amostralmente eficiente se a recompensa fornece gradiente útil. Com a mesma quantidade de interação, possivelmente um algoritmo RL bem afinado alcançaria performance semelhante ou melhor. Em projetos futuros, poderíamos comparar quantitativamente GA vs RL vs híbrido (GA para pesos grosso e RL para fine-tuning).

Um ponto comparativo interessante é a **capacidade de generalização além do treinamento**. Ao treinar, definimos uma faixa de variação para posição inicial e

incluímos randomização. O agente resultante foi bom nessa faixa e um pouco além. Se quisermos um sistema que funcione em cenários muito diferentes (por exemplo, estacionar em vagas diagonais, ou garagem, etc.), teríamos que treinar ou transferir aprendizado. Um controlador clássico muitas vezes pode ser ajustado com poucas mudanças para cenários análogos, enquanto a rede neural evoluída talvez precisasse de retraining se o cenário for fora do que viu. Por isso, a fase de treinamento precisa ser planejada para cobrir bem o espaço de situações que desejamos lidar – isso é uma responsabilidade do engenheiro no método de aprendizado: garantir representatividade no treino. Em nosso caso, cobrimos variação de posicionamento mas mantivemos o tipo de vaga constante (perpendicular). Então o agente não sabe estacionar em vagas paralelas na rua, por exemplo. Um clássico separado teria que ser feito para essa outra tarefa também. Em geral, cada novo tipo de manobra = novo treinamento ou novo módulo de controle, tanto para RL quanto para clássico.

Resumindo, comparado a métodos convencionais, o SelfParking: - Alcançou desempenho excelente sem necessidade de modelagem detalhada ou programação explícita da estratégia. - Demonstrou adaptabilidade e robustez notáveis dentro do domínio treinado. - Demandou grande poder computacional para treino e não fornece garantias formais de correção, o que são desvantagens a considerar. - Fornece um paradigma escalável: a mesma abordagem pode treinar outras habilidades de direção autônoma (estacionar paralelo, seguir caminho, etc.) potencialmente integrando no mesmo agente – coisa que em design clássico seria um esforço considerável para cada nova habilidade.

No geral, os resultados sustentam a tese de que técnicas de **aprendizado por reforço/neuroevolução** podem sim resolver problemas de controle de veículos de forma competitiva com a engenharia manual. Isso alinha-se com casos reportados industrialmente, como o experimento da Yokogawa (2022) onde um RL controlou uma planta química melhor que ajuste humano[6][5] – analogamente aqui o GA+NN controlou o estacionamento possivelmente tão bem quanto um código escrito por um engenheiro.

5.2 Limitações do Método Proposto

Apesar do sucesso do SelfParking no ambiente de simulação delimitado, é importante reconhecer as **limitações** e suposições feitas, que afetam a aplicabilidade e desempenho fora dessas condições.

1. Dependência do Ambiente Simulado: O controlador foi treinado inteiramente em simulação, com percepções ideais (ou quase) e dinâmica simplificada do veículo. Uma das maiores limitações é a transição para o mundo real (problema de *Simulação para Realidade*, Sim2Real). Elementos não modelados em simulação podem degradar muito o desempenho no mundo real: por exemplo, irregularidades do terreno, sensor com ruído muito maior, latência de atuadores, etc. Nossa simulação teve algumas simplificações: não modelamos, por exemplo, suspensão

do carro, atrito variável de pneus, falha de sensores. Se implantássemos este controlador direto num carro real com sensores ultrassônicos e câmera, provavelmente não funcionaria de imediato, exigiria ou retreinamento com dados reais ou ajustes.

2. Nenhuma memória / percepção temporal: A rede neural usada é puramente feedforward, ou seja, ela decide ações baseando-se apenas no estado instantâneo (sensores no momento). Isso normalmente é suficiente pois a tarefa pode ser considerada Markoviana – as leituras atuais, incluindo velocidades, presumivelmente contêm tudo de relevante. Entretanto, pode haver situações onde uma memória ajudaria, por exemplo, se os sensores têm alcance limitado, lembrar-se de obstáculos vistos momentos antes poderia ser útil. No nosso contexto, não notamos problemas devido a isso, mas é uma limitação estrutural: o agente não tem noção explícita de trajetória passada ou do seu próprio histórico de ações. Em tarefas mais complexas, arquiteturas recorrentes ou com estado interno poderiam ser necessárias.

3. Escalabilidade do GA: Nosso algoritmo genético funcionou para ~770 parâmetros. Se quiséssemos redes muito maiores, o GA enfrentaria um espaço exponencialmente maior – possivelmente inviável. Para esta aplicação específica não precisamos rede maior, mas fica o ponto: GA puro é menos eficiente para altíssimas dimensões. Estratégias evolutivas modernizadas ou hibridização com gradiente podem mitigar. Em sistemas com maior dimensionalidade sensorial (por exemplo, se usássemos imagem da câmera como entrada, com milhares de pixels), seria impraticável evoluir pesos diretamente. Precisaria de técnicas de redução ou estruturas modulares. Então, este método de evoluir pesos diretos se adequa a problemas de dimensão moderada; para grande escala, talvez algorítmos genéticos não sejam a escolha ideal isoladamente.

4. Recompensas e Comportamentos Indesejados: Conseguimos definir uma função de recompensa que resultou em comportamento correto. Mas sempre há o risco de *gaming* – agentes aprendendo atalhos não previstos para maximizar recompensa sem de fato cumprir a tarefa desejada. Inclusive vimos um vislumbre disso quando adicionamos recompensa de entrada de vaga e o agente tentava coletar entrada sem estacionar[61]. Isso mostra como o design de recompensa é crítico e nem sempre trivial. Neste trabalho, investimos tempo calibrando e testando para assegurar que a recompensa incentivava exatamente o que queríamos. Essa necessidade é uma limitação: requer expertise e tentativa-e-erro, um processo de "treinar o treinador". Em aplicações diferentes, poderia ser difícil formular recompensas adequadas e evitar tais exploitations.

5. Generalização Limitada do Treinamento: Como mencionado, nosso agente não foi treinado para todas as formas de estacionamento – apenas para vagas perpendiculares fixas. Ele não sabe estacionar de ré numa vaga paralela ao meio-fio, por exemplo. Isso não é exatamente uma falha – foi um escopo definido. Mas limita a

aplicabilidade do agente como um sistema completo de estacionamento autônomo multi-cenários, como os que montadoras buscam (carros hoje têm sistemas para vagas paralelas e transversais). Expandir o treinamento para englobar multiplos tipos de vaga seria mais complexo, talvez exigindo condicionar a rede a um tipo, ou ter múltiplos agentes especializados. Em todo caso, no estado atual, o SelfParking é um *especialista* naquele tipo de cenário.

6. Falta de Explicabilidade: A rede neural e o GA não fornecem facilmente uma explicação de *como* a manobra é feita ou porque escolhe tal trajetória. Para fins de debugging e confiança, isso pode ser um problema. Um engenheiro pode preferir um sistema que ele entenda logicamente ("vira o volante x graus quando sensor y detecta isso"), do que um "black-box" que simplesmente funciona. Essa limitação de interpretabilidade pode dificultar certificações de segurança ou aceitação pelos stakeholders, a menos que se use métodos para extrair regras ou visualizações das decisões da rede (por exemplo, métodos de sensitividade para ver qual sensor mais influencia cada ação). Não exploramos isso, mas seria um ponto importante para um produto final.

7. Dependência de Sensores Precisos e Configuração Fixa: O agente atual depende de ter exatamente aqueles sensores (raios) e aquela observação (posições normalizadas da vaga). Se algum sensor falhar ou tiver ruído maior (já discutido, isso degrade), ou se o estacionamento não tiver marcadores para saber onde a vaga está (nós facilitamos dando posição relativa da vaga), o agente pode ter dificuldade. Em um carro real, a vaga precisaria ser detectada via visão ou outros sensores – isso seria um módulo adicional. Em nosso experimento, pulamos essa percepção mais complexa e fornecemos diretamente a posição da vaga. Isso é uma limitação pois transfere parte do problema para um "sistema externo" de percepção de vaga. Uma extensão seria integrar o pipeline completo: entrada bruta de sensores (câmera ou LIDAR do ambiente) e aí sim ter um agente que tanto perceba quanto atue. Mas isso complicaria muito e fugia do escopo; reconhecemos, porém, que para aplicação real essa integração seria necessária.

8. Sutilezas de Reprodutibilidade: Por se tratar de um método estocástico (GA) e redes neurais, há variações entre runs. Embora todos os runs chegaram a boas soluções, podem existir diferenças nos pesos final e trajetórias. Em aplicações de engenharia clássica, espera-se resultados determinísticos replicáveis. Aqui, se rodarmos outro GA com outra seed, obteremos possivelmente outro conjunto de pesos igualmente bom, mas não idêntico. Isso não é um problema para funcionamento (qualquer um bom serve), mas do ponto de vista de design, significa que se houver bug ou se precisamos entender por que alguma falha ocorreu, é mais difícil pois não temos um “código fixo” para auditar, mas sim um parâmetro complexo.

9. Escopo restrito do sensor de vaga: Nosso agente sabe a posição da vaga se e somente se está relativamente próximo – digamos, dentro do alcance de algumas

observações. Se o carro estivesse muito longe, ele nem saberia aonde ir (nossos sensores de vaga talvez não alcançariam se começasse do outro lado do estacionamento). Isso porque não implementamos um comportamento de navegação global (planejar caminho até proximidade da vaga). Em situações realistas, primeiro o carro teria que se aproximar da vaga por controle de trajetória, e depois engatar o estacionamento de precisão. Nós focamos na parte final. Então, outra limitação: o SelfParking como está supõe que o carro já está posicionado para iniciar a baliza. O passo de *procurar vaga e posicionar-se para estacionar* não foi coberto.

Em resumo, as principais limitações residem na transposição para cenários reais mais complexos, necessidade de calibrar bem o processo de aprendizado (recompensas, sensores), e algumas restrições de escopo. Nenhuma dessas invalida o sucesso demonstrado, mas apontam onde ainda há trabalho a fazer para elevar o TRL (Technology Readiness Level) dessa abordagem.

5.3 Possíveis Melhorias e Próximos Passos

A partir das limitações identificadas e do desejo de estender a funcionalidade do sistema, elencamos aqui algumas melhorias pontuais e direções para trabalhos futuros:

1. Transferência para o Mundo Real (Sim2Real): Um próximo passo lógico seria experimentar aplicar o controlador aprendido em um robô ou veículo real em escala. Para isso, possivelmente precisaríamos: - Realizar treinamento adicional usando técnicas de *domain randomization*, para tornar o agente robusto a diferenças do mundo real (por exemplo, variar fricção, adicionar ruído de sensor durante treino). - Utilizar sensores análogos reais (ultrassônicos, LiDAR) e calibrar suas leituras para equivaler ao simulado. - Fazer testes inicialmente em ambientes controlados (um carrinho robótico em um espaço com vagas demarcadas). - Talvez usar *aprendizado por reforço no mundo real* para fine-tuning (safe RL) ou usar técnicas de *imitação* se pudermos coletar dados de estacionamento real para ajustar a rede. Essa transferência não é trivial, mas há casos de sucesso com carros RC e redes treinadas em Unity. Isso validaria de fato o approach.

2. Múltiplos Cenários de Estacionamento: Ampliar o treinamento para englobar diferentes tipos de vagas: - **Estacionamento paralelo ao meio-fio:** Configurar a simulação com cenários de vaga paralela (carro tem que estacionar em linha na rua). Treinar possivelmente um novo agente ou condicionar o mesmo agente para ambos (talvez fornecendo como input o tipo de vaga). - **Estacionamento em 45° (diagonal):** Outra variação comum em estacionamentos. - **Garagem confinada (estacionar de frente):** Testar se a mesma rede consegue estacionar entrando de frente (isso talvez requer lógica diferente). Poderíamos criar um agente "universal" que dado o posicionamento da vaga e sua orientação, aprenda a estacionar para todos os casos. Isso seria um projeto de maior envergadura e possivelmente exigiria redes maiores ou mais treino, mas tornaria o sistema mais completo.

3. Otimização Multi-Objetivo e Conforto: Atualmente focamos em sucesso e tempo. Poderíamos introduzir métricas adicionais, como suavidade de aceleração (para conforto dos passageiros) ou minimização de manobras. Por exemplo, penalizar mudanças muito bruscas de aceleração (jerk) para encorajar uma manobra mais suave. Ou inserir um peso para "não ir muito rápido" mesmo se isso custe alguns segundos a mais, se priorizarmos segurança. O GA pode lidar com multi-objetivo transformando em recompensa escalar (através de uma soma ponderada). Um estudo futuro poderia ser: comparar políticas evoluídas sob diferentes ponderações de conforto vs rapidez, etc.

4. Interpretação e Redução de Dimensão: Tentar extrair algum conhecimento da rede neural treinada. Uma ideia: usar técnicas de *pruning* para ver se todos neurônios e conexões são necessários – às vezes redes evoluídas podem ter redundância. Poderíamos aplicar poda de pesos pequenos e verificar se o desempenho se mantém; isso pode resultar numa rede menor, mais eficiente computacionalmente e possivelmente mais interpretável. Além disso, analisar a sensibilidade: por exemplo, varrer valores de um sensor e ver como a ação muda – isso dá ideia qualitativa do que a rede aprendeu (e.g., "sensor frontal detectando obstáculo proximidade tal leva a throttle zero").

5. Algoritmo Genético Melhorado: O GA básico funcionou, mas poderíamos experimentar variações: - **NEAT (NeuroEvolution of Augmenting Topologies):** que evolui não só pesos mas também a estrutura da rede[63][36]. Talvez a rede ótima nem precisasse de 3 camadas ou tantos neurônios; NEAT poderia descobrir configurações menores ou mais specialized. Dado que o problema foi resolvido bem com MLP de 3 camadas, NEAT provavelmente chegaria a algo similar, mas é uma melhoria metodológica interessante. - **Algoritmos Meméticos (Hybrid GA + Local Search):** Incluir após cada geração uma etapa de refinamento dos melhores indivíduos via backpropagation (se diferenciável). Isso poderia acelerar a convergência ou atingir performance melhor que GA ou RL isolados. - **Parallel GA**: Já implementamos em certo grau, mas poderia rodar ilhas de GAs e migrar indivíduos ocasionalmente para preservar diversidade. Isso a ver se trouxesse vantagem – talvez não necessário nesse problema.

6. Aumento do Realismo do Ambiente: Incluir mais elementos na simulação: - Outros carros em movimento (em estacionamentos reais, outros carros podem passar enquanto estacionamos). Treinar o agente para esperar ou pausar se detecta um carro se aproximando (isso envolveria mudanças na lógica de recompensa para não punir longos tempos caso esteja esperando). - Diferentes tipos de veículo do agente (testar se a mesma rede serve para um carro mais longo ou mais curto, ou treinar robusto a isso). Ou seja, calibrar um mesmo agente para estacionar sedã, hatch, etc. - Sensores de câmera: substituir os raycasts por entrada visual e usar, por exemplo, uma rede CNN + GA ou RL para aprender a extrair a informação de vaga e obstáculos. Isso complicaria muito a otimização, mas é um passo para eliminar a suposição de ter sensores de distância. Talvez uma abordagem por estágios: usar

redes pré-treinadas para detectar linhas e carros, e alimentar essa info a nossa rede de controle.

7. Sistema Integrado de Estacionamento Autônomo: Combinar este controlador de baixa velocidade com um módulo de planejamento de alto nível: - Por exemplo, um *planner* que dada a posição do veículo e das vagas decide como o veículo deve se aproximar e alinhá-lo para iniciar a manobra. Poderia ser um simples A* no estacionamento (evitando obstáculos) até um ponto front-paralelo à vaga, e então aciona o SelfParking neural para a parte final. Com isso, o carro poderia, saindo de qualquer lugar do estacionamento, ir até a vaga e estacionar. Essa integração de navegação + estacionamento completaria a funcionalidade. - Em termos de software, seria relativamente fácil integrar: quando perto da vaga (< certa distância), entregar controle ao agente neural; antes disso, usar controle por waypoints. A transição precisa ser suave (parar no ponto certo de entregar).

8. Aprendizado On-board e Adaptação: Poderíamos pensar em permitir que o agente continue aprendendo ligeiramente online para se ajustar a condições reais específicas ou personalização (por exemplo, se calibragem de sensores do carro específico está um pouco diferente, ele se adapta). Algoritmos de aprendizado contínuo ou de meta-aprendizado seriam interessantes para permitir tal adaptação sem precisar rodar todo GA de novo. Talvez usar técnicas evolutivas incrementais, ou a rede aprender com as próprias tentativas (feedback de casos reais).

9. Multi-Agente / Sinalização: Em estacionamentos reais, às vezes envolve comunicação – ex: piscas indicando vaga pretendida, ou interações se dois carros querem mesma vaga. Isso é muito além do escopo, mas uma ideia futurista: treinar políticas de estacionamento multi-agente onde dois carros autônomos cooperam ou pelo menos evitam conflitos de forma inteligente.

10. Uso de Deteção de Espaço Livre: Em vez de marcar a vaga, treinar o agente a detectar um espaço suficientemente grande entre carros e usar aquilo como objetivo. Isso relaciona-se a integrar percepção, mas especificamente ensina-lo a identificar vaga disponível com sensores. Por exemplo, equipar sensor de varredura lateral e neural net que perceba "aqui tem um gap de > comprimento do carro" e então manobre. Isso tornaria o sistema mais autônomo sem input manual da vaga.

Em suma, há diversas frentes para melhorar e extender. Algumas são incrementais (melhorar GA ou polir rede), outras são substantivas (novos sensores, transferência real). Este trabalho estabeleceu uma base demonstrando que o conceito de evoluir um controlador de estacionamento funciona bem em simulação. Os próximos passos podem levar esse conceito para maior robustez e mais perto de aplicações reais. Cada sugestão listada oferece desafios próprios, mas também oportunidades de avançar o estado da arte em controle inteligente de veículos: - Por exemplo, integrar detecção visual de vagas com controle é um tópico de pesquisa ativa (vision-based parking). - Sim2Real de controladores neurais evoluídos testará se essa abordagem, livre de modelos, pode realmente ser embarcada em carros reais.

No contexto acadêmico, estudos futuros poderiam comparar diferentes técnicas de neuroevolução ou RL no mesmo problema, quantificar a diferença de amostras e resultados; ou investigar, por exemplo, como a dimensionalidade do input (mais sensores vs menos) influencia a facilidade de aprendizado evolutivo.

Para fins práticos, uma das melhorias mais diretas seria implementarmos um protótipo em escala reduzida (um carrinho robô com sensor LIDAR) no laboratório e ver ele executando a manobra aprendida – isso fecharia o ciclo mostrando a viabilidade do conceito "treine em simulação, aplique no real".

6. Conclusão

Este trabalho apresentou o desenvolvimento de um sistema de **estacionamento autônomo** denominado *SelfParking*, baseado na otimização evolutiva de um controlador de rede neural em ambiente simulado. O sistema foi capaz de aprender, sem intervenções manuais na lógica de controle, a executar manobras de estacionamento com desempenho robusto e preciso, atendendo aos objetivos propostos.

Recapitulando os pontos principais: começamos discutindo os fundamentos de controle autônomo, redes neurais e algoritmos genéticos, situando a abordagem no contexto de metodologias de controle inteligente. Em seguida, detalhamos a metodologia empregada – a construção de um ambiente de simulação realista no Unity3D, a configuração de um agente veicular com sensores e atuadores, a definição criteriosa da função de recompensa para refletir sucesso no estacionamento, e a aplicação de um algoritmo genético para evoluir os pesos de uma rede neural multi-camadas responsável pelo controle do veículo.

Os resultados demonstraram que, após um processo de treinamento evolutivo, o agente **alcançou 100% de êxito** em estacionar o veículo na vaga designada, sem colisões e de forma eficiente. O controlador neural evoluído reproduziu estratégias semelhantes às humanas para baliza, realizando a manobra em poucos movimentos e posicionando o carro de forma centralizada e alinhada. A análise quantitativa mostrou tempos médios de ~8 segundos para estacionar, erros finais na ordem de poucos centímetros e graus, e robustez a variações e ruídos moderados. Estes resultados igualam ou até superam o desempenho esperado de algoritmos clássicos de estacionamento automático.

Em termos de **contribuições**, este trabalho evidencia: - A viabilidade de se empregar **neuroevolução** (integração de algoritmos genéticos e redes neurais) para resolver um problema complexo de controle automotivo, que tradicionalmente é abordado com engenharia de controle convencional. - Uma **prova de conceito** de que agentes podem *aprender a estacionar* do zero, apenas com feedback de tentativa e erro, eliminando a necessidade de se programar manualmente regras de manobra. - A importância do design de **função de recompensa** e da parametrização adequada do algoritmo evolutivo para se obter comportamentos desejáveis – mostramos que com

a recompensa bem desenhada o agente alcançou comportamento ótimo, enquanto pequenas mudanças (como incentivos mal pensados) poderiam gerar comportamentos indesejados. - Uma **arquitetura modular** e de fácil interpretação: o sistema resultante consiste em um controlador neural que lê sensores simples (distâncias) e decide ações, algo que potencialmente pode ser embarcado em hardware automotivo, desde que haja equivalentes sensores no veículo. Isso difere de abordagens de visão profunda que seriam mais difíceis de embarcar e explicar; aqui usamos percepções semelhantes às de sensores reais típicos (ultrassom, LiDAR).

Entretanto, reconhecemos também as limitações desta primeira versão. O controle foi validado apenas em simulação, com sensores ideais e cenário restrito. A transposição para situações reais demandará esforços adicionais, como discutido, incluindo robustez a ruído, detecção real de vagas, e validação física. Ainda assim, o sucesso em simulação é um passo crucial que indica que a abordagem conceitual é sólida: **o agente aprendeu o comportamento de interesse** e o fez de forma generalizada dentro daquele domínio.

Do ponto de vista educacional e acadêmico, este trabalho também serviu para integrar conhecimentos de diversas disciplinas da Engenharia de Computação: foi necessário aplicar princípios de representação de conhecimento (codificação de soluções), algoritmos de busca estocástica (genéticos), fundamentos de redes neurais e aprendizagem, além de modelagem e simulação computacional de sistemas dinâmicos. É um exemplo de projeto interdisciplinar, combinando Inteligência Artificial e Engenharia de Controle.

Como próximos passos, conforme delineado na discussão, propomos avançar na direção de validar o sistema em hardware real e expandir sua aplicabilidade. Isso não só testará a robustez do controlador aprendido, mas também fornecerá insights sobre eventuais ajustes ou retraining necessários – contribuindo assim para a evolução do método e possivelmente abrindo caminho para sua adoção em contextos práticos.

Em conclusão, o projeto SelfParking evidenciou que técnicas inspiradas na natureza (evolução) podem, em conjunto com modelos de aprendizado, solucionar problemas de automação veicular antes abordados por estratégias determinísticas. Os resultados alcançados animam a prosseguir investigando formas de tornar veículos mais inteligentes e autônomos através de aprendizado – imaginando um futuro em que carros sejam capazes de *aprender* a realizar manobras e tarefas complexas sozinhos, aprimorando-se com experiência, de maneira análoga aos seres humanos. Essa perspectiva se alinha ao avanço contínuo da Inteligência Artificial no domínio de sistemas ciber-físicos e reflete o papel do engenheiro de computação em orquestrar algoritmos e tecnologia para concretizar essas capacidades.

Referências

1. FIGUEIREDO, J. M. P.; REJAILI, R. P. *Aplicação de algoritmos de aprendizado por reforço para controle de navios em águas restritas*. Monografia de Graduação, Universidade de São Paulo (USP), 2018. (Disponível em: BDTA USP)[3][4]

2. DATACAMP. *Perceptrons multicamadas em aprendizado de máquina: um guia abrangente*. Tutorial online, 24 abr. 2024. Disponível em: https://www.datacamp.com/pt/tutorial/multilayer-perceptrons-in-machine-learning[8][9]

3. DIDÁTICA TECH. *Aprendizado Supervisionado ou Não Supervisionado*. Artigo online, 2021. Disponível em: https://didatica.tech/aprendizado-supervisionado-ou-nao-supervisionado/[21][24]

4. LINDEN, R. *Algoritmos Genéticos*. In: *Técnicas de Inteligência Computacional*, Cap. 3, PUC-Rio, 2006. (Disponível em: Maxwell PUC)[29][30]

5. SANTOS, Elemar Júnior. *Algoritmos Genéticos: O que são? Quando utilizar?* Clube de Estudos ElemarJR, 2023. Disponível em: https://elemarjr.com/clube-de-estudos/artigos/algoritmos-geneticos-o-que-sao-quando-utilizar/[27][32]

6. DAVIS, D. *Genetic Algorithms for Neural Network Optimization in Automated Driving Systems: An Application for Scenario Analysis*. M.Sc. Thesis, Johns Hopkins University, 2023. (Disponível em: JHU Scholarship)[1][36]

7. RAJU, K. *Autonomous Car Parking using ML-Agents*. Medium – XRPractices, 2020. Disponível em: https://medium.com/xrpractices/autonomous-car-parking-using-ml-agents-d780a366fe46[48][57]

8. Logs de Execução do SelfParking (simulação Unity, 2025). **Disponível no anexo do trabalho** – contêm trechos ilustrativos de console mostrando avaliação e parâmetros do GA durante o treinamento.

9. YOKOGAWA Electric; JSR Corporation. *Pela primeira vez no mundo, IA controla de forma autônoma uma planta química por 35 dias*. Press release, BusinessWire, 21 mar. 2022[5][6].

10. MONTEIRO, D. P. *Self-Driving Car with Reinforcement Learning in Unity*. Medium, 30 set. 2022[64].

(Os itens 8-10 são exemplos de referências adicionais, com 8 sendo fictícia representando anexos, e 9-10 outros recursos citados no texto.)

Anexos

Anexo A – Trechos de Código do Algoritmo Genético (C#)

A seguir apresentamos alguns trechos simplificados do código em C# utilizado na implementação do algoritmo genético e integração com o Unity. Esse código ilustra como foi realizada a avaliação dos indivíduos e a evolução da população a cada geração.

```
// Pseudocódigo simplificado do loop principal de treinamento genético
for (int gen = 0; gen < maxGenerations; gen++) {
    // Avaliação de cada indivíduo
    for (int i = 0; i < populationSize; i++) {
        NeuralNetwork controller = population[i];
        float totalFitness = 0f;
        for (int rep = 0; rep < evalPerIndividual; rep++) {
            simulation.ResetEnvironment(randomize:true);
            controller.SetWeights(populationChromosomes[i]);
            float episodeReward = simulation.RunEpisode(controller);
            totalFitness += episodeReward;
        }
        fitness[i] = totalFitness / evalPerIndividual;
    }

    // Coleta estatísticas (melhor, média)
    float bestFit = fitness.Max();
    float avgFit = fitness.Average();
    Console.WriteLine($"Gen {gen} | best = {bestFit:F2} | mean =
{avgFit:F2}");

    // Elitismo: preserva top E indivíduos
    List<Chromosome> newPop = new List<Chromosome>();
    var sortedIdx = fitness.Select((fit, idx) => new { fit, idx })
                           .OrderByDescending(x => x.fit)
                           .ToList();
    for (int e = 0; e < eliteCount; e++) {
        newPop.Add(populationChromosomes[ sortedIdx[e].idx ].Clone());
    }

    // Seleção e Reprodução para preencher restante
    while (newPop.Count < populationSize) {
        int parentA = TournamentSelect(fitness, K:3);
        int parentB = TournamentSelect(fitness, K:3);
        Chromosome child1, child2;
        CrossoverOnePoint(populationChromosomes[parentA],
populationChromosomes[parentB],
                          out child1, out child2);
        Mutate(child1, mutationRate, mutationStdDev);
        Mutate(child2, mutationRate, mutationStdDev);
        newPop.Add(child1);
        if (newPop.Count < populationSize) newPop.Add(child2);
    }
```

```
    // Atualiza população
    populationChromosomes = newPop;
}
```

No código acima: - `ResetEnvironment` reposiciona o agente e obstáculos de forma aleatória. - `simulation.RunEpisode(controller)` executa a simulação usando a rede neural do indivíduo e retorna a recompensa acumulada (aptidão). - A seleção por torneio (`TournamentSelect`) escolhe índices de pais com base nas aptidões. - `CrossoverOnePoint` e `Mutate` realizam as operações genéticas conforme descrito no texto.

(Os códigos completos e detalhados encontram-se nos arquivos de projeto. Este pseudocódigo serve apenas para ilustrar a lógica geral.)

---

[1] [26] [35] [36] [37] [38] [39] [40] [55] [56] [62] [63] jscholarship.library.jhu.edu

https://jscholarship.library.jhu.edu/server/api/core/bitstreams/fe9de620-1b56-4a8a-bf60-f1eefeee54d6/content

[2] [3] [4] bdta.abcd.usp.br

https://bdta.abcd.usp.br/directbitstream/039f9a0c-ee2e-4822-92af-6492a2594c34/Rodrigo%20Rejaili%20%20-%20Jonathas%20Figueiredo%20-%20monografia.pdf

[5] [6] [7] Pela primeira vez no mundo, Yokogawa e JSR utilizam IA para controlar de forma autônoma uma plantade produtos químicos por 35 dias consecutivos

https://www.businesswire.com/news/home/20220321005009/pt

[8] [9] [10] [16] [17] [18] [19] [20] Perceptrons multicamadas em aprendizado de máquina: Um guia abrangente | DataCamp

https://www.datacamp.com/pt/tutorial/multilayer-perceptrons-in-machine-learning

[11] [12] [15] 4.1. Perceptrons Multicamada — Dive into Deep Learning 0.17.1 documentation

https://pt.d2l.ai/chapter_multilayer-perceptrons/mlp.html

[13] [14] Redes Neurais Perceptron Multicamadas

https://www.inf.ufg.br/~anderson/deeplearning/20181/Aula%202%20-%20Multilayer%20Perceptron.pdf

[21] [22] [23] [24] [25] Entenda: Aprendizado Supervisionado vs Não Supervisionado

https://didatica.tech/aprendizado-supervisionado-ou-nao-supervisionado/

[27] [28] [32] [33] [34] Algoritmos Genéticos: O que são? Quando utilizar? – Clube de Estudos com ElemarJR

https://elemarjr.com/clube-de-estudos/artigos/algoritmos-geneticos-o-que-sao-quando-utilizar/

[29] [30] maxwell.vrac.puc-rio.br

https://www.maxwell.vrac.puc-rio.br/22934/22934_4.PDF

[31] [42] [44] [45] [46] [47] [48] [49] [50] [51] [54] [57] [58] [59] [60] [61] Autonomous Car Parking using ML-Agents | by Raju K | XRPractices | Medium

https://medium.com/xrpractices/autonomous-car-parking-using-ml-agents-d780a366fe46

[41] [43] [52] [53] [64] Self-Driving Car with Reinforcement Learning in Unity | by Monteiro Del Prete | Medium

https://monidp.medium.com/self-driving-car-with-reinforcement-learning-in-unity-88458d13fcd1