Trabalho de Conclusão de Curso

Direção autônoma com aprendizado por reforço e *TinyML*

Valério Nogueira Rodrigues Júnior

Orientador:

Prof. Dr. Erick de Andrade Barboza

Maceió, 04 de Dezembro de 2024

Valério Nogueira Rodrigues Júnior

Direção autônoma com aprendizado por reforço e *TinyML*

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Engenharia de Computação do Instituto de Computação da Universidade Federal de Alagoas.

Orientador:

Prof. Dr. Erick de Andrade Barboza

Maceió, 04 de Dezembro de 2024

Catalogação na fonte
Universidade Federal de Alagoas
Biblioteca Central
Divisão de Tratamento Técnico
Bibliotecária: Helena Cristina Pimentel do Vale – CRB4/661

F866c Rodrigues Junior, Valério Nogueira.
Direção autônoma com aprendizado por reforço e TinyML / Valério Nogueira Rodrigues Júnior. – 2025.
38 f.: il.

Orientador: Erick de Andrade Barboza.
Monografia (Trabalho de Conclusão de Curso em Engenharia de Computação) – Universidade Federal de Alagoas, Instituto de Computação. Maceió, 2024.

Bibliografia: f. 37-38.

1. Inteligência artificial. 2. Sistemas embarcados. 3. Aprendizado por reforço. 4. TinyML 5. Deep Q-Learning. I. Título.

CDU: 004.8

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Engenharia de Computação do Instituto de Computação da Universidade Federal de Alagoas, aprovada pela comissão examinadora que abaixo assina.

Documento assinado digitalmente
gov.br
ERICK DE ANDRADE BARBOZA
Data: 06/12/2024 08:49:04-0300
Verifique em https://validar.iti.gov.br

Prof. Dr. Erick de Andrade Barboza - Orientador
Instituto de Computação
Universidade Federal de Alagoas

Documento assinado digitalmente
gov.br
GLAUBER RODRIGUES LEITE
Data: 07/12/2024 09:14:47-0300
Verifique em https://validar.iti.gov.br

Prof. Dr. Glauber Rodrigues Leite - Examinador
Instituto de Computação
Universidade Federal de Alagoas

Documento assinado digitalmente
gov.br
ICARO BEZERRA QUEIROZ DE ARAUJO
Data: 06/12/2024 14:40:18-0300
Verifique em https://validar.iti.gov.br

Prof. Dr. Ícaro Bezerra Queiroz de Araújo - Examinador
Instituto de Computação
Universidade Federal de Alagoas

Maceió, 04 de Dezembro de 2024

Resumo

O aprendizado de máquina desempenha um papel importante no desenvolvimento de veículos autônomos. Este trabalho tem como objetivo abordar o problema direção autônoma utilizando apenas imagens em um protótipo de mini-veículo de baixo custo. Para isso, foram empregados algoritmos de aprendizado por reforço para treinar o modelo de inteligência artificial em um ambiente simulado virtualmente e posteriormente técnicas de *TinyML* para otimizá-lo e torná-lo compatível para ser executado em microcontroladores. Como resultado foi obtido um modelo capaz de dirigir o veículo corretamente em uma pista. Além disso, constatou-se uma redução significativa na latência do sistema com a abordagem de *TinyML* se comparado com a abordagem convencional de computação em nuvem.

Palavras-chave: inteligência artificial, sistemas embarcados, aprendizado por reforço, *TinyML*, direção autônoma, *Deep Q-Learning*.

Abstract

Machine learning plays an important role in the development of autonomous vehicles. This work aims to tackle the autonomous driving problem using only images in a low-cost mini-vehicle prototype. For this, reinforcement learning algorithms were employed to train the artificial intelligence model in a simulated virtual environment and posteriorly TinyML techniques were used to make it compatible to be executed by microcontrollers. This resulted in a model capable of successfully drive the vehicle in a road and a comparison between the conventional cloud based and TinyML approaches showed a significant reduction in the system latency.

Keywords: artificial intelligence, embedded systems, reinforcement learning, *TinyML*, autonomous driving, *Deep Q-Learning*.

Sumário

Lista de Figuras v
Lista de Tabelas vi

1 Introdução **1**
1.1 Introdução 1
1.2 Tabalhos relacionados 2
1.3 Objetivos 3
1.3.1 Objetivos gerais 3
1.3.2 Objetivos específicos 3

2 Fundamentação Teórica **4**
2.1 Aprendizado de máquina 4
2.2 Aprendizado por reforço 5
2.2.1 Processo de Decisão de Markov 5
2.2.2 Exemplo: Come-come 7
2.2.3 *Q-Learning* 8
2.3 Redes neurais artificiais 10
2.3.1 Aprendizado profundo 12
2.3.2 Redes neurais convolucionais 12
2.4 *Deep Q-Learning* 14
2.5 *TinyML* 14
2.5.1 Benefícios do *TinyML* 14
2.5.2 Técnicas de otimização 16
2.6 Direção autônoma 17
2.6.1 Direção autônoma em mini-veículos 19

3 Metodologia **20**
3.1 Simulação 20
3.2 Treinamento e validação 25
3.3 Implantação do modelo embarcado 26

4 Resultados **30**
4.1 Treinamento 30
4.2 Implantação 32
4.3 Síntese dos Resultados 33

5 Conclusão **35**
5.1 Desafios e Soluções 35
5.2 Contribuições do Projeto 36
5.3 Trabalhos Futuros 36

Lista de Figuras

1 Ciclo de interação entre o agente e o ambiente no aprendizado por reforço. No instante t o agente observa o estado s_t e executa a ação a_t no ambiente. Este, por sua vez, devolve um novo estado s_{t+1} e uma recompensa r_{t+1}. 6
2 Um exemplo de conFiguração inicial do jogo "Come-come" 7
3 Propagação do erro das estimativas no *Q-Learning* 10
4 Modelo do *perceptron* 12
5 Exemplo de uma um modelo baseado na *LeNet* com duas camadas convolucionais e uma camada densa 13
6 Exemplos de estado obtidos na simulação 20
7 Captura de tela da simulação implementada, exibindo o ambiente, o agente e a câmera virtual presa ao veículo, que serve como o meio utilizado por ele para observar o estado do ambiente. 21
8 Representação geométrica do modelo do mapa 22
9 Mapeamento das coordenadas de textura de um segmento de rua de comprimento L. 23
10 Diagrama demonstrando a recompensa obtida pelo agente. A linha tracejada representa a linha de referência da pista. No lado esquerdo, o veículo dirige na contramão e por isso em 10 passos receberá -1 no total. Já no lado direito, o veículo dirige na faixa correta e receberá 1 25
11 Captura de tela da ferramenta *Tensorboard*. Nela é possível visualizar as métricas capturadas durante os treinamentos em tempo real. 27
12 *Kit* de desenvolvimento *T-Camera S3* da *LILYGO*. 27
13 Etapas utilizadas na conversão do modelo *Pytorch* em um modelo do *Tensorflow Lite* 28
14 Retorno médio durante o treinamento. 31
15 Duração média do episódio durante o treinamento. 31
16 Comparação entre o retorno obtido com o modelo original e com o modelo otimizado em 100 episódios. 32
17 Retorno obtido pelo agente na avaliação durante o treinamento. 33
18 Tempo de inferência ao decorrer de um episódio com 100 passos. 34

Lista de Tabelas

2.1 Características de cada nível de automação de sistemas de direção em relação aos papéis do motorista humano e do sistema de direção. 18
2.2 Comparativo entre o hardware de uma *Raspberry Pi 3B+* e o de uma *Raspberry Pi Pico*. 19
3.1 Espaço de ações do agente 24
3.2 Arquitetura da rede utilizada para treinar o agente 26
4.1 Espaço ocupado pelos modelos e taxa de compressão em relação ao original. . 32
4.2 Tempo de inferência das duas abordagens de implantação do modelo. 33

1

Introdução

1.1 Introdução

O paradigma de *Internet* das coisas (*IoT*) revolucionou a maneira com que as pessoas interagem com dispositivos eletrônicos. Estes dispositivos, que antes só conseguiam realizar ações específicas e pré programadas, agora podem adaptar-se ao usuário através da coleta de dados de sensores, como câmeras, microfones, sensores de temperatura entre outros e de modelos de aprendizado de máquina.

A utilização desses modelos geralmente requer um alto poder computacional. Para reduzir os custos de produção desses dispositivos muitas empresas optam por utilizar uma abordagem de computação em nuvem. Assim, o dispositivo fica encarregado de coletar os dados através de seus sensores e enviá-los para um servidor pela *Internet*. Estes servidores processam esses dados e retornam o resultado para o dispositivo.

No entanto, questões envolvendo a privacidade dos usuários, o custo de manutenção de uma infraestrutura em nuvem e latência do sistema são proibitivas para tarefas que necessitam de uma resposta rápida. Com o aumento expressivo do poder computacional dos microcontroladores surge a alternativa do *TinyML*, que promete mover a computação da nuvem para dentro dos próprios dispositivos, alcançado através da otimização dos modelos. Dessa maneira, as tarefas realizadas pelos dispositivos podem ser cada vez mais complexas.

Além disso, com o desenvolvimento do aprendizado profundo, houve um ressurgimento do interesse no aprendizado por reforço, visto que os algoritmos existentes só podiam ser utilizados em problemas pequenos. As redes neurais profundas permitiram a utilização desses algoritmos em problemas com o espaço de estados exponencialmente maior. Assim é possível obter modelos que aprendem através da interação com o ambiente, sem a necessidade de uma base de dados rotulada ou uma implementação específica.

O desenvolvimento de carros autônomos também foi uma área bastante beneficiada com o avanço do aprendizado de máquina. Veículos autônomos representam aumento de segurança no

trânsito pois podem evitar acidentes, redução nos congestionamentos pois podem permitir que os veículos fiquem mais próximos sem que entrem em colisão e também representam uma maior eficiência no uso de recursos, permitindo que duas pessoas compartilhem o mesmo veículo e ele faça sozinho o percurso entre os dois condutores, por exemplo.

O controle de um carro robótico envolve o processamento de dados provenientes de diversos sensores em um ambiente altamente dinâmico e não estruturado. Além disso, uma ação tomada por um carro autônomo podem ter consequências prolongadas. Por isso, técnicas como aprendizado por reforço podem oferecer uma excelente alternativa, pois podem aprender a maximizar a recompensa a longo prazo.

Atualmente é possível aplicar métodos que conseguem realizar as tarefas relativas à direção autônoma, como manter o veículo dentro da faixa, utilizando apenas sensores de baixo custo, como câmeras convencionais. No entanto, o estudo desses métodos ainda é restrito visto que para realizar os experimentos ainda são necessários equipamentos de custo elevado, como o próprio veículo.

Para contornar este problema é comum utilizar robôs móveis simulando os veículos, normalmente controladas por comandos enviados através da rede por um computador. Com o aumento do poder computacional dos microcontroladores, tornou-se possível executar modelos de redes neurais artificiais nesses dispositivos. Isso traz a possibilidade de estudar os métodos de direção autônoma nos mesmos robôs, porém mantendo o processamento inteiramente no robô.

Desta forma, este trabalho pretende abordar o problema da direção autônoma utilizando aprendizado por reforço desde a simulação de um ambiente até a aplicação do resultado obtido no mundo real, utilizando *TinyML*.

1.2 Tabalhos relacionados

PAN et al. (2017) utilizou uma abordagem de aprendizado por reforço com a técnica de destilação de conhecimento para fazer com que um modelo reduzido conseguisse imitar um modelo maior. Para implantar o modelo, foi utilizado um hardware compatível com um computador de mesa . Já BECHTEL et al. (2018) utilizou uma placa *Raspberry3 Pi 3* para executar um modelo treinado com aprendizado supervisionado que utiliza apenas a imagem de uma única câmera e posteriormente também aplicou a técnica em uma *Raspeberry Pi Pico* (BECHTEL; WENG; YUN, 2022).

1.3 Objetivos

1.3.1 Objetivos gerais

Demonstrar a viabilidade da aplicação de modelos de inteligência artificial em microcontroladores utilizando aprendizado por reforço, desde o treinamento do modelo até a implantação, para a realização da tarefa de direção autônoma. O modelo final deve ser capaz de guiar um veículo corretamente numa pista respeitando a contramão, utilizando apenas imagens como entrada.

1.3.2 Objetivos específicos

- Implementar simulação de veículo autônomo em ambiente virtual 3D.
- Treinar modelo capaz de realizar a tarefa de dirigir o veículo corretamente no ambiente simulado.
- Utilizar técnicas de *TinyML* para otimizar o modelo obtido e implantá-lo em um microcontrolador.
- Comparar a performance das abordagens de *TinyML* e computação em nuvem com relação à latência do sistema.

2

Fundamentação Teórica

Neste capítulo, serão definidos conceitos básicos de aprendizado de máquina, com foco no aprendizado por reforço. A Seção 2.1 apresenta os principais tipos de aprendizado de máquina. A Seção 2.2 introduz os conceitos básicos de aprendizado por reforço, da modelagem com processos de decisão de Markov até o algoritmo *Q-Learning*. A Seção 2.3 mostra os fundamentos das redes neurais artificiais, com ênfase em aprendizado profundo e nas redes neurais convolucionais. Já a Seção 2.4 fala sobre a versão "profunda" do *Q-Learning*, o *Deep Q-Learning*. Na Seção 2.5 é apresentado o conceito de *TinyML* e seus benefícios e desafios. Por fim, a Seção 2.6 discorre sobre os principais desafios e limitações do problema da direção autônoma, com foco no estudo de algoritmos em versões reduzidas de veículos.

2.1 Aprendizado de máquina

O aprendizado de máquina é uma subárea da inteligência artificial que estuda os algoritmos capazes de obter modelos matemáticos a partir de dados, sem a necessidade de uma implementação explícita. Ela se divide em três paradigmas principais, sendo eles:

- **Aprendizado supervisionado**: O aprendizado supervisionado é a forma mais comum de aprendizado de máquina (LECUN; BENGIO; HINTON, 2015). Nessa categoria os algoritmos utilizam uma espécie de supervisor que fornece pares entrada-saída desejados. Assim, o algoritmo consegue ajustar os parâmetros do modelo com base na diferença entre a saída correta (supervisor) e a saída predita. Normalmente é utilizado em problemas de classificação ou regressão e o supervisor consiste em bases de dados rotulados previamente.

- **Aprendizado não-supervisionado**: Como o nome indica, os algoritmos não-supervisionados não necessitam de um supervisor. Ao invés de tentar predizer um valor ou categoria, esses algoritmos tentam encontrar padrões nos dados. Dessa forma fica a cargo do usuário

interpretar o resultado obtido. Os exemplos mais comuns de uso desses métodos são os problemas de agrupamento, onde deseja-se separar as amostras da base de dados em grupos com características similares, sem definir *a priori* estes grupos.

- **Aprendizado por reforço**: A principal característica desse tipo de algoritmo é a interação de um agente com um ambiente. Essa interação pode gerar um valor de recompensa para o agente, possivelmente futura, que serve para ajustar os parâmetros do modelo reforçando ou corrigindo seu comportamento.

2.2 Aprendizado por reforço

Como dito anteriormente, o aprendizado por reforço se baseia na interação entre um agente e um ambiente. Essa interação é ilustrada no diagrama da Figura 1. Nela, é possível ver o ciclo que se inicia com o agente fazendo uma leitura do ambiente e obtendo para si o estado inicial s_0. A partir disso, em cada instante t, aplica uma ação a_t no ambiente que faz com que este, por sua vez, devolva um novo estado s_{t+1} acompanhado de uma recompensa r_{t+1}.

O **retorno** R_T é definido como o total de recompensas obtido pelo agente em T iterações a partir do instante t, $R_T = r_{t+1} + r_{t+2} + \ldots + r_{t+T}$. Nesse caso, T é a duração de um **episódio**. Porém existem problemas em que não se tem um episódio definido. Para impedir que esse valor cresça indefinidamente nessas situações geralmente é utilizado um coeficiente de amortização $\gamma \in [0,1]$:

$$R_T = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}. \quad (1)$$

Variando o valor de γ é possível controlar a contribuição das recompensas futuras. Dessa forma, quanto mais distante menor será sua relevância para a tomada de decisão. No caso extremo, onde $\gamma = 0$, apenas a recompensa atual é levada em consideração.

Dessa forma, uma característica marcante do aprendizado por reforço é a recompensa atrasada. É possível que uma determinada ação não gere uma recompensa ao ser escolhida, porém pode ser necessário para que no futuro o agente receba uma recompensa maior. Isso deve ser levado em conta para que o agente consiga aprender a realizar a tarefa da melhor maneira possível.

2.2.1 Processo de Decisão de Markov

Para modelar o problema do aprendizado por reforço é utilizado o conceito de Processo de Decisão de Markov (PDM). Um PDM é caracterizado por:

- Um conjunto de estados S
- Um conjunto de ações A

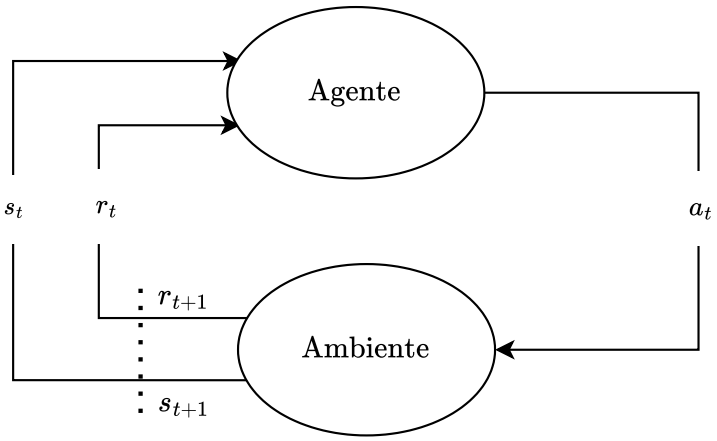

Figura 1: Ciclo de interação entre o agente e o ambiente no aprendizado por reforço. No instante t o agente observa o estado s_t e executa a ação a_t no ambiente. Este, por sua vez, devolve um novo estado s_{t+1} e uma recompensa r_{t+1}.
Fonte: Autor.

- Uma função $P_a(s,s')$ que representa a probabilidade de sair do estado s para o estado s' através da ação a
- Uma função $R_a(s,s')$ que representa a recompensa obtida ao sair do estado s para o estado s' através da ação a

Além disso é necessário obedecer à propriedade de Markov, que diz que as transições futuras só dependem do estado atual.

Assim, ”aprender por reforço” significa interagir com o ambiente em busca de um objetivo enquanto tenta maximizar o sinal de retorno. O comportamento que o agente segue para isso é chamado de **política**. A política π de um agente é uma função que mapeia um par estado-ação para um número real que representa a probabilidade do agente tomar a ação a dado que ele se encontra no estado s. A política ótima π^* é a política que consegue resolver o problema de forma ótima, isto é, obtendo o maior retorno possível a partir de qualquer estado.

Outro conceito importante no estudo do aprendizado por reforço é a definição das funções **valor-estado** e **valor-ação**. Essas funções expressam o retorno esperado seguindo a política π.

A função valor-estado representa o valor esperado do retorno dado que o agente se encontra no estado s pode ser escrita como

$$V^{\pi}(s) = E_{\pi}\{R_T|s_t = s\} \tag{2}$$

enquanto a função valor-ação é definida pelo valor esperado do retorno dado que o agente se encontra no estado s e toma a ação a naquele instante pode ser escrita como

$$Q^{\pi}(s,a) = E_{\pi}\{R_T|s_t = s, a_t = a\}. \tag{3}$$

Juntas, essas duas funções fornecem meios de avaliar e comparar quantitativamente estados ou ações.

2.2.2 Exemplo: Come-come

Para exemplificar os conceitos acima considere o cenário de uma versão modificada do jogo eletrônico *Pac-man*. Nela, o objetivo é coletar todos os pontos em um labirinto $n \times m$ com e inimigos. O jogo termina quando todos os pontos são coletados ou quando o jogador se encontra na mesma posição que um inimigo. O jogador inicia na posição $(0,0)$ e os inimigos são posicionados aleatoriamente no tabuleiro. Exceto a posição inicial do jogador, todas as outras possuem um ponto em cada. A Figura 2 ilustra a conFiguração inicial de uma partida para $n = 4$, $m = 5$ e $e = 3$.

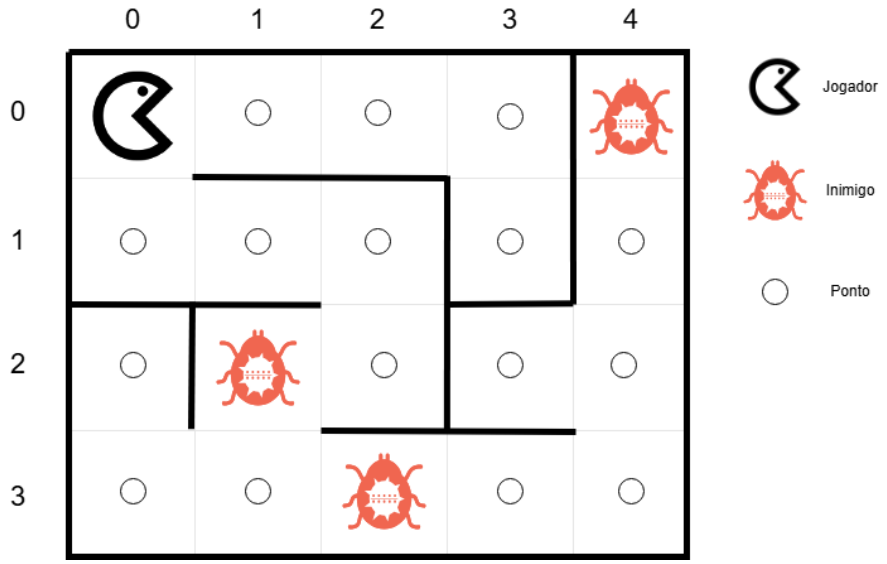

Figura 2: Um exemplo de conFiguração inicial do jogo ”Come-come”
Fonte: Autor.

Em cada passo o jogador deve escolher uma posição adjacente à sua para se mover. Se a nova posição do jogador contiver um ponto, a pontuação do jogador é incrementada em uma unidade e o ponto daquela posição é removido.

Além da definição do ambiente, é necessário definir o estado, as ações e a função de recompensa. Para simplificar o exemplo, considere que o agente só consegue enxergar as 4 posições adjacentes à sua. Seja $P = \{\text{Inválido}, \text{Parede}, \text{Ponto}, \text{Visitado}, \text{Inimigo}\}$ o conjunto de possíveis tipos para cada posição, significando:

- **Inválido**: Indica uma posição fora do tabuleiro.
- **Parede**: Há uma parede entre a posição atual do jogador e esta posição.
- **Ponto**: Há um ponto na posição e não há um inimigo.

- **Visitado**: Indica que não há inimigo nesta posição e que o jogador já passou por ela anteriormente.
- **Inimigo**: Há um inimigo nesta posição.

Seja $\mathrm{T}(i,j) : \mathbb{N}^2 \mapsto P$ a função que retorna o tipo da posição (i,j) do tabuleiro. Assim, dado que o jogador se encontra na posição (i,j) o estado é o conjunto $\{T(i,j+1), T(i+1,j), T(i,j-1), T(i-1,j)\}$. A partir da análise combinatória, o total de estados possíveis é $5 \times 5 \times 5 \times 5 = 625$. Esse número cresce exponencialmente de acordo com o tamanho do campo de visão do jogador. Tomando como exemplo a Figura 2 o estado obtido pelo agente é dado por $\{\text{Ponto}, \text{Ponto}, \text{Inválido}, \text{Inválido}\}$.

O conjunto de ações disponíveis é dado por $A = \{\text{Direita}, \text{Esquerda}, \text{Para cima}, \text{Para baixo}\}$ e representam a direção desejada do movimento. Caso haja uma parede entre a posição atual e a posição desejada o efeito da ação é nulo. O mesmo acontece caso a posição desejada esteja fora do tabuleiro. Por fim, a recompensa será definida através da pontuação obtida após a execução da ação.

2.2.3 *Q-Learning*

O algoritmo *Q-Learning* (WATKINS, 1989) foi um dos primeiros grandes avanços no estudo do aprendizado por reforço (SUTTON; BARTO, 2018). Ele é um método de diferença temporal que utiliza uma abordagem gulosa para a escolha das ações.

O conceito fundamental para o *Q-Learning*, como o próprio nome indica, é a função valor-ação $Q(s,a)$. Ela é representada através de uma tabela que é indexada de acordo com as dimensões dos estados e ações, chamada tabela Q. Quando as variáveis são contínuas elas normalmente elas passam por uma etapa de discretização.

O aprendizado consiste em comparar duas estimativas diferentes de $Q(s_t,a_t)$. A primeira é representada pela tabela Q, que contém os retornos esperado para todos os pares estado-ação do problema. A segunda estimativa é obtida através da decomposição de $Q(s_t,a_t)$ em termos da recompensa atual e do valor da ação seguinte:

$$\begin{aligned} Q(s_t,a_t) &= E\{R_T | s = s_t, a = a_t\} \\ &= E\{r_{t+1} + r_{t+2} + r_{t+3} + \ldots + r_{t+T} | s = s_t, a = a_t\} \\ &= E\{r_{t+1} | s = s_t, a = a_t\} + E\{r_{t+2} + r_{t+3} + \ldots + r_{t+T} | s = s_{t+1}, a = a_{t+1}\} \\ &= E\{r_{t+1} | s = s_t, a = a_t\} + Q(s_{t+1}, a_{t+1}). \end{aligned} \tag{4}$$

Essa equação indica que, no instante t, o retorno esperado para o par estado-ação é igual à recompensa esperada somada ao valor esperado para o par estado-ação no instante $t+1$. Assim, substituindo $E\{r_{t+1}\}$ pelo valor de r_{t+1} dado pelo ambiente tem-se a segunda estimativa

$$Q'(s_t,a_t) = r_{t+1} + Q(s_{t+1}, a_{t+1}). \tag{5}$$

Pelo fato de utilizar um valor concreto para r_{t+1}, Q' se aproxima mais do valor real. Para calcular o termo $Q(s_{t+1}, a_{t+1})$ são avaliadas todas as ações possíveis no estado s_{t+1}, isto é, a tabela Q é consultada para toda ação $a \in A$ e o maior valor obtido é escolhido. Por esse motivo, o *Q-Learning* é considerado um algoritmo *off-policy*, ou seja, não segue a política. Dessa forma, a diferença entre as duas estimativas é utilizada para atualizar a tabela Q:

$$\begin{aligned} Q(s_t, a_t) &\leftarrow Q(s_t, a_t) + \alpha[Q'(s_t, a_t) - Q(s_t, a_t)] \\ &\leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_{a \in A} Q(s_{t+1}, a) - Q(s_t, a_t)] \end{aligned} \tag{6}$$

onde α é a taxa de aprendizagem.

O Algoritmo 1 apresenta o pseudocódigo do *Q-Learning*.

Algorithm 1 *Q-Learning*

```
Inicializa a tabela Q com valores arbitrários
for episódio = 1, M do
    Reinicia ambiente
    Observa o estado inicial s_0
    s ← s_0
    for t = 1, T do
        Escolhe a ação a
        Executa a ação a no ambiente e observa próximo estado s_{t+1} e recompensa r_{t+1}
        if s_{t+1} é terminal then
            r ← r_{t+1}
        else
            r ← r_{t+1} + γ max_{a'∈A} Q(s_{t+1}, a')
        end if
        Q(s_t, a_t) ← Q(s_t, a_t) + α[r − Q(s_t, a_t)]
        s_t ← s_{t+1}
    end for
end for
```

Para exemplificar o que foi apresentado, considere um problema episódico com duração de T passos onde $r_i = 0$ para $i < T$ e $r_T \neq 0$ e que todos os valores da tabela Q foram inicializados com 0. Durante o primeiro episódio, o sinal de erro $Q'(s_t, a_t) - Q(s_t, a_t)$ será sempre igual a 0 e os valores da tabela Q não sofrerão alterações. Quando o agente chegar no penúltimo estado, s_{T-1}, ele receberá a recompensa r_T. Seguindo a regra de atualização da tabela Q, isso fará com que o sinal de erro agora também seja diferente de 0. Dessa forma, o valor de $Q(s_{T-1}, a_{T-1})$ será atualizado.

A Figura 3 mostra como esse erro é propagado entre os estados durante o aprendizado. Nela, os estados são representados por círculos e as ações pelas setas. Quanto mais escuro for o círculo, maior é o retorno esperado naquele estado. Considere que no segundo episódio o agente chega no estado s_{T-2}. Dessa vez, $Q(s_{T-1}, a_{T-1}) \neq 0$ e portanto $Q(s_{T-2}, a_{T-2})$ também será atualizado.

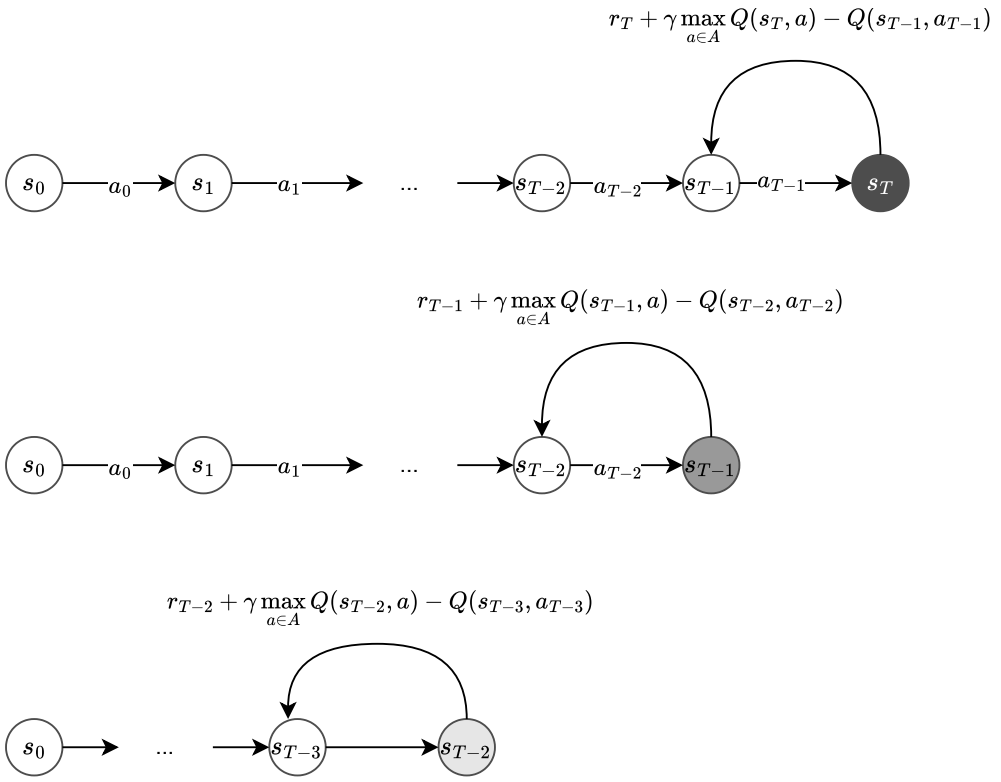

Figura 3: Propagação do erro das estimativas no *Q-Learning*
Fonte: Autor.

Após um certo número de iterações, é esperado que esse sinal finalmente chegue ao estado inicial s_0. O agente, que inicialmente não tinha nenhuma informação sobre o ambiente e agia aleatoriamente, agora consegue agir com base na experiência adquirida.

Existem ainda diferentes variações que alteram o funcionamento do algoritmo. Uma delas é o modo como a tabela Q é inicializada (SUTTON; BARTO, 2018). Ela pode ser inicializada de maneira "otimista", isto é, com valores positivos. Desse modo, inicialmente todas as ações parecerão boas alternativas. No entanto, caso o agente não receba uma recompensa esse valor será atualizado com um valor menor. Assim, essa estratégia faz com que o agente priorize as ações ainda não tentadas.

2.3 Redes neurais artificiais

Uma rede neural consiste em um conjunto de células chamadas neurônios que conseguem se associar e receber/transmitir estímulos através de reações bioquímicas e impulsos elétricos. Um neurônio pode se conectar a outro através de estruturas conhecidas como dendritos. Essas conexões são chamadas de sinapses e permitem a passagem de informação entre eles.

Ao receber um certo grau de estímulo, como o de um órgão sensorial, um neurônio pode transmiti-lo para os demais conectados a ele. A associação de milhões dessas células é o que

permite que o sistema nervoso dos animais processe os estímulos obtidos através dos sentidos, desencadeando um comportamento em resposta. Além disso, a associação dos neurônios não é estática: sinapses podem ser criadas, fortalecidas ou desfeitas, de acordo com a experiência adquirida pelo ser vivo. Dessa forma, é como se o comportamento do animal fosse um mapeamento entre um padrão de estímulos e uma resposta do organismo, como uma contração muscular, por exemplo.

Tentando explicar como a experiência é armazenada no cérebro, (ROSENBLATT, 1958) desenvolveu o primeiro algoritmo de aprendizado de máquina, chamado *perceptron*. Matematicamente ele pode ser descrito por

$$y = \sigma(\sum w_i x_i + b) \tag{7}$$

onde y é a saída, σ a função de ativação, w_i e b são os pesos e x_i as entradas. A Figura 4 mostra visualmente a estrutura do *perceptron*.

É possível traçar uma analogia direta entre o neurônio. As entradas x_i correspondem aos estímulos recebidos, a saída y ao estímulo transmitido e os pesos w_i e b representam as sinapses. A função de ativação serve para mapear o valor acumulado e simular o comportamento de neurônios biológicos, que podem "disparar"ou não dependendo do estímulo.

O algoritmo de treinamento do *perceptron* é um caso de aprendizado supervisionado. A partir da diferença entre a saída obtida e a saída esperada de uma amostra conhecida os pesos são atualizados seguindo as regras

$$w_i \leftarrow w_i + \eta(t-y) * x_i \tag{8}$$

e

$$b \leftarrow b + \eta(t-y) \tag{9}$$

onde η é a taxa de aprendizagem, t é a saída esperada e y a saída obtida. Esse processo é repetido até que ele consiga predizer corretamente todas as amostras utilizadas para o treinamento.

De maneira geral, o *perceptron* só funciona casos os dados em questão sejam linearmente separáveis. Observando o argumento da função de ativação na equação 7, é possível perceber que ela é equivalente à equação que define um hiperplano no $\mathbb{R}^n$, onde n é a quantidade de dimensões da entrada $\mathbf{x}$. Assim, o que ele avalia é se o ponto em questão está de um lado ou do outro do hiperplano, através do sinal do resultado.

Assim como nas redes neurais biológicas, o poder das redes neurais artificiais está na possibilidade de arranjar uma quantidade obscena de neurônios. O método mais comum de organização consiste em dispor vários neurônios em camadas, e utilizar as saídas dos neurônios de uma camada como a entrada para os neurônios da camada posterior. Essa estrutura é conhecida como *perceptron* multicamadas. Dessa forma, é como se cada *perceptron* trabalhasse em separar diferentes dimensões dos dados, e esse resultado fosse combinado ao final, tornando

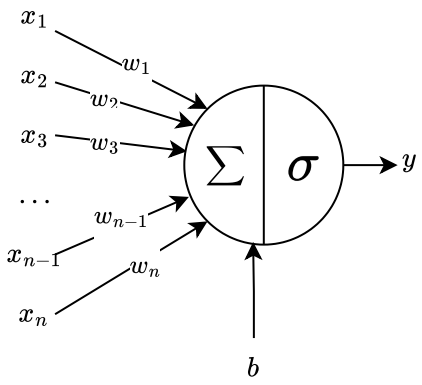

Figura 4: Modelo do *perceptron*
Fonte: Autor.

possível trabalhar com dados que não são linearmente separáveis.

2.3.1 Aprendizado profundo

Com o avanço do poder computacional disponível foi possível construir redes neurais artificiais cada vez mais complexas. Isso inaugurou um novo subcampo do aprendizado de máquina, o aprendizado profundo. A principal característica do aprendizado profundo é a quantidade de camadas utilizadas nas redes.

Para que fosse viável treinar redes com tantos parâmetros foi necessário o desenvolvimento de um algoritmo específico para fazer o cálculo dos gradientes de forma eficiente. Esse algoritmo, chamado de retropropagação, faz o caminho contrário do erro, atualizando primeiramente os pesos das útlimas camadas. Com isso, ele reaproveita os resultados calculados para ajustar os pesos das camadas anteriores.

2.3.2 Redes neurais convolucionais

Redes neurais convolucionais são redes neurais que possuem camadas convolucionais. Essas camadas funcionam através da operação de convolução de sinais. Experimentos em gatos revelaram que no córtex cerebral dos felinos há neurônios que só respondem a estímulos em regiões específicas da retina (HUBEL; WIESEL et al., 1959). Inspirado nisso, as camadas convolucionais realizam as operações levando em consideração toda a região em volta de cada *pixel*.

Camadas convolucionais utilizam um número muito inferior de parâmetros quando comparado com uma arquitetura densa. A quantidade de parâmetros contidos em uma camada convolucional não depende do tamanho da entrada, mas da quantidade de filtros desejada e o tamanho dos mesmos. Considere, por exemplo, que os *pixels* de uma imagem $n \times m$ são reorga-

nizados com um vetor de dimensão $n * m$ e utilizados como entrada para um MLP. Desse modo, seriam necessários $n * m$ pesos para cada neurônio da camada seguinte. Isso torna impraticável utilizar essa arquitetura para tratar imagens. No entanto, ao utilizar uma camada convolucional com f filtros $k_x \times k_y$, só é necessário armazenar pesos para os $f * k_x * k_y$ componentes dos filtros. Normalmente os valores k_x e k_y são pequenos, o que reduz drasticamente o tamanho do modelo.

O processo de extração de características das imagens é semelhante ao que acontece nos animais. No caminho que o estímulo visual percorre da retina até o cérebro, chamado de via visual, ele passa por neurônios especializados para reagir de forma diferente a depender do tipo do estímulo. Alguns neurônios podem ser mais sensíveis a linhas, outros a padrões específicas. Isso também acontece nas redes convolucionais, onde as primeiras camadas aprendem características mais abstratas, como formas geométricas, enquanto as camadas mais profundas aprendem características mais específicas, como texturas ou objetos.

A Figura 5 mostra a arquitetura *LeNet* (LECUN; BENGIO; HINTON, 2015). Ela foi criada para realizar a classificação de dígitos escritos à mão. Na época, foi a primeira vez em que o algoritmo de retropropagação foi utilizado para otimizar os filtros das camadas convolucionais. Seu sucesso fez com que se tornasse o modelo padrão para processamento de imagens com aprendizado profundo. Ela pode ser separada em duas partes: um extrator de características e um classificador (ou regressor).

O extrator de características é composto pelas camadas convolucionais, organizadas sequencialmente. Ele recebe como entrada a imagem e tem como saída os mapas de características extraídos, que são redimensionados para um vetor unidimensional no final do processo. Para reduzir a dimensão desse vetor, as camadas convolucionais podem ser intercaladas com camadas de *pooling*. Elas geram uma imagem menor em que cada *pixel* é o resultado da aplicação de uma métrica no *pixels* de uma região da imagem de entrada.

A segunda parte da rede consiste em um MLP convencional, que pode ser tanto para um problema de classificação como de regressão. Ele é utilizado para classificar as caracteristicas extraídas nas camadas anteriores.

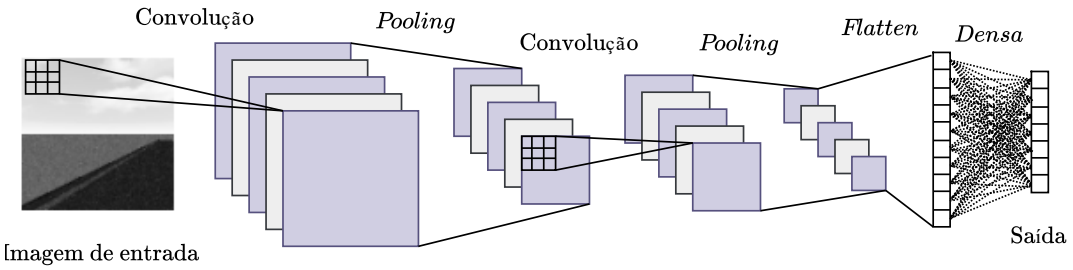

Figura 5: Exemplo de uma um modelo baseado na *LeNet* com duas camadas convolucionais e uma camada densa

Fonte: Autor.

2.4 *Deep Q-Learning*

Deep Q-Learning é um algoritmo de aprendizado por reforço combinado com aprendizado profundo. A ideia do algoritmo é utilizar uma rede neural para aproximar a função $Q(s,a)$. Com isso é possível utilizar representações de estado com muitas dimensões, como imagens. O algoritmo consiste de um *buffer* de experiência, chamado aqui de *replay buffer* e uma rede neural que recebe como entrada o estado do agente e tem como saída $Q(s,a')\forall a' \in A$, chamada de rede Q.

O *replay buffer* é utilizado para armazenar as transições executadas durante o treinamento. As atualizações dos pesos da rede são feitas a partir de amostras coletadas dele. Isso faz com que o processo de treinamento seja mais suave, pois utilizar transições consecutivas é ineficiente devido a forte correlação entre as amostras (MNIH et al., 2013).

A função de custo $L_i(\theta_i)$ representa a função de custo no instante i para a rede com parâmetros θ_i. Ela é semelhante à regra de atualização do *Q-Learning* e pode ser escrita como

$$L_i(_i) = E\{(y_i - Q(s,a;\theta_i))^2\} \tag{10}$$

onde

$$y_i = E[r + \gamma \max_{a' \in A} Q(s',a';\theta_{i1})|s,a] \tag{11}$$

é o valor alvo para a iteração i. A cada iteração do agente com o ambiente uma amostra é coletada do *replay buffer* e o gradiente descendente é utilizado para atualizar a rede. O Algoritmo 2 apresenta o pseudocódigo do *Deep Q-Learning*.

2.5 *TinyML*

Nesse contexto surge o *TinyML* que em tradução livre pode ser entendido como "aprendizado de máquina minúsculo". Segundo (ABADADE et al., 2023), o *TinyML* foca na implantação de modelos de aprendizado de máquina otimizados em dispositivos pequenos e de baixo consumo como microcontroladores à bateria e sistemas embarcados. Dentre os principais benefícios de sua utilização é possível citar a redução na latência, maior privacidade e baixo consumo de energia.

2.5.1 Benefícios do *TinyML*

Apesar dos serviços de computação em nuvem possuírem um poder computacional muito maior, a transmissão de dados entre o dispositivo e o servidor representa um gargalo. Fatores como a disponibilidade de rede ou o tipo de conexão que o dispositivo possui afetam diretamente a performance do sistema. Assim, a latência de um modelo de *TinyML* é significativamente menor que um modelo sendo executado em um servidor (NOVAC et al., 2021).

Algorithm 2 *Deep Q-Learning*

Inicialize o *replay buffer* $\mathcal{D}$ com tamanho N
Inicialize a rede Q com pesos aleatórios
for episódio $= 1, M$ **do**
    Reinicie o ambiente
    Observe o estado inicial s_0
    $s \leftarrow s_0$
    for $t = 1, T$ **do**
        Escolha a ação a
        Execute a ação a e observa próximo estado s_{t+1} e recompensa r_{t+1}
        Armazene a transição $(s_t, a_t, r_{t+1}, s_{t+1})$ no *replay buffer*
        if s_{t+1} é terminal **then**
            $r \leftarrow r_{t+1}$
        else
            $r \leftarrow r_{t+1} + \gamma \max_{a' \in A} Q(s_{t+1}, a')$
        end if
        Colete aleatoriamente um lote de amostras $(s_t, a_t, r_{t+1}, s_{t+1})$ do *replay buffer*
        Defina

$$y_j = \begin{cases} r_j & \text{se } s_{j+1} \text{ é terminal} \\ r_j + \gamma \max_{a' \in A} Q(s_{j+1}, a') & \text{caso contrário} \end{cases}$$

        Utilize $[y_j - Q(s_j, a_j; \theta_j)]^2$ para atualizar os pesos da rede Q com o gradiente descendente
        $s_t \leftarrow s_{t+1}$
    end for
end for

Uma das maiores vantagens de utilizar *TinyML* ao invés de uma arquitetura mais convencional de computação em nuvem é não necessitar de uma conexão com a *Internet* (NOVAC et al., 2021). No caso de sistemas embarcados, muitas vezes não é possível adicionar conectividade ao dispositivo pois pode encarecer o produto final. Além disso, existem situações em que não há disponibilidade de rede como aplicações envolvendo sensoriamento em ambientes remotos.

Questões de privacidade e segurança dos dados são de fundamental importância no contexto de dispositivos inteligentes. Invasão de privacidade, uso indevido e vazamentos de dados são apenas alguns dos problemas que podem ocorrer quando se utiliza um dispositivo conectado à *Internet*. No caso de dispositivos inteligentes, como *smartphones* e vestíveis, essa preocupação é ainda maior pois nestes dispositivos estão presentes diversos sensores que podem coletar dados sensíveis do usuário. Dessa forma, o uso de *TinyML* se apresenta uma solução que resolve a maior parte desse problemas pois os dados são gerados e consumidos no próprio dispositivo (NOVAC et al., 2021), além de diminuir a desconfiança do usuário final.

O consumo de energia também é bastante reduzido quando se utiliza um modelo *TinyML*. Enquanto uma abordagem de computação na nuvem pode chegar a consumir 1000 W de potência, microcontroladores operam na faixa de 25 a 300 mW (ABADADE et al., 2023). Além disso, as tecnologias de comunicação utilizadas para conectar o dispositivo à *Internet* aumentam esse consumo. Portanto, o *TinyML* permite executar modelos de inteligência artificial em dispositivos com grandes restrições de energia, como aqueles alimentados por baterias.

2.5.2 Técnicas de otimização

Apesar da evolução do *hardware* e barateamento do custo de produção levando ao aumento de memória e poder de processamento, microcontroladores ainda são dispositivos com baixíssimo poder computacional se comparados a computadores convencionais. Para extrair o máximo de performance nesses dispositivos, forma desenvolvidas diversas técnicas de otimização de modelo. Dentre as principais técnicas é possível citar (ABADADE et al., 2023):

- **Poda**: A poda consiste em remover do modelo os pesos que possuem pouca contribuição para o resultado final da rede (HAN; XIAO; LI, 2024). Há diferentes tipos de poda, que podem ir de pesos individuais até camadas inteiras. No caso da poda de pesos ao final do treinamento é definindo um patamar mínimo para os pesos e aqueles que estiverem abaixo desse valor são desconsiderados. Já na poda de neurônios, são desconsiderados todos os pesos relacionados a esse neurônio. Em camadas convolucionais é possível realizar a poda nos filtros, removendo os filtros que obtiverem menor contribuição. No caso de poda de camadas uma técnica mais sofisticada é utilizada. Essa técnica, conhecida como destilação de conhecimento, consiste em treinar o modelo completo e em seguida utilizar esse modelo como supervisor para o treinamento de uma versão reduzida desse modelo. Desse modo o objetivo é fazer com que a versão menor aprenda a copiar o comportamento da maior.

- **Quantização**: A quantização é utilizada para reduzir o espaço ocupado pelos pesos e acelerar o processo de inferência utilizando tipos de dados mais eficientes (NOVAC et al., 2021). Normalmente os pesos são representados por números de ponto flutuante de 32 ou 64 *bits*. Após a quantização, esses valores podem ser armazenados em tipos bem menores, como inteiros de 8 *bits*. Além de ocuparem 4 vezes menos espaço as operações realizadas nesses tipos, como adição ou multiplicação, são mais eficientes do que em números de ponto flutuante. A conversão dos valores é feita analisando a distribuição dos valores presentes na rede para determinar os parâmetros a fim de minimizar a perda de acurácia ocasionada pela perda de precisão.

- **Fatoração de posto baixo**: Essa técnica consiste em fatorar as matrizes de pesos em duas matrizes menores, onde é possível reconstruir a matriz original através da multiplicação das duas (SWAMINATHAN et al., 2020). Dessa maneira só é necessário armazenar essas duas matrizes menores no modelo, que pode reduzir significativamente o espaço total.

- **Compressão**: Métodos de compressão sem perda são utilizados para reduzir o espaço ocupado na memória do microcontrolador. Um exemplo de algoritmo que pode ser utilizado é a codificação de Huffman (HUFFMAN, 1952). Ela associa a cada ”letra” uma sequência binária de tamanho inversamente proporcional à sua frequência em um texto. Desse modo o símbolo que mais se repete é substituído pela sequência de menor tamanho, que geralmente possui apenas alguns *bits*.

2.6 Direção autônoma

Em 2014, a Sociedade de Engenharia Automotiva (SAE) publicou o padrão *SAE International J3016* que define a *tarefa de direção dinâmica* como todos os aspectos operacionais e táticos referentes à direção de um veículo. Alguns exemplos desses aspectos são o controle da direção, aceleração e desaceleração, resposta a eventos etc. Além disso, o documento também define e descreve níveis de automação em que um sistema de direção pode ser categorizado a fim de simplificar a comunicação e facilitar a colaboração entre os setores técnico e político (INTERNATIONAL, 2014).

A seguir são descritos os níveis de automação possíveis, com seus respectivos nomes e descrições. A Tabela 2.1 detalha as características de cada nível em relação aos papéis do motorista humano e do sistema de direção autônomo.

0. **Sem automação**: Desempenho integral realizado pelo motorista humano de todos os aspectos da tarefa de direção dinâmica, mesmo quando aprimorado por aviso ou sistemas de intervenção.

1. **Assistência ao motorista**: A execução de um modo de direção específico realizado por um sistema de assistência ao motorista da direção ou aceleração/desaceleração usando

informação sobre o ambiente de direção com a expectativa de que o motorista humano desempenhe os aspectos restantes da tarefa de direção dinâmica.

2. **Automação parcial**: A execução de um modo de direção específico realizado por um ou mais sistemas de assistência ao motorista da direção e aceleração/desaceleração usando informação sobre o ambiente de direção com a expectativa de que o motorista humano desempenhe os aspectos restantes da tarefa de direção dinâmica.

3. **Automação condicional**: Desempenho de um modo de direção específico realizado por um sistema de direção autônomo de todos os aspectos da tarefa de direção dinâmica com a expectativa de que o motorista humano responderá apropriadamente uma solicitação de intervenção.

4. **Alta automação**: Desempenho de um modo de direção específico realizado por um sistema de direção autônomo de todos os aspectos da tarefa de direção dinâmica, mesmo se um motorista humano não responde apropriadamente uma solicitação de intervenção.

5. **Automação completa**: Desempenho integral realizado por um sistema de direção autônomo de todos os aspectos da tarefa de direção dinâmica sob todas as condições ambientais e de estrada que podem ser tratadas por um motorista humano.

Nível SAE	Nome	Direção e aceleração/-desaceleração	Monitoramento do ambiente de direção	Desempenho alternativo da tarefa de direção dinâmica	Modos de direção
Motorista humano monitora o ambiente de direção					
0	Sem automação	Humano	Humano	Humano	Nenhum
1	Assistência ao motorisa	Humano e sistema	Humano	Humano	Alguns
2	Automação parcial	Sistema	Humano	Humano	Alguns
Sistema de direção autônomo (”sistema”) monitora o ambiente de direção					
3	Automação condicional	Sistema	Sistema	Humano	Alguns
4	Alta automação	Sistema	Sistema	Sistema	Alguns
5	Automação completa	Sistema	Sistema	Sistema	Todos

Tabela 2.1: Características de cada nível de automação de sistemas de direção em relação aos papéis do motorista humano e do sistema de direção.

2.6.1 Direção autônoma em mini-veículos

Carros autônomos são sistemas de tomada de decisão autônomos que processam fluxos de observações vindas de diferentes fontes, como câmeras, radares, *LiDAR*[1], sensores ultrassônicos, *GPS*[2] e/ou sensores inerciais (GRIGORESCU et al., 2020). Geralmente esses veículos são construídos tendo como base um carro convencional, com estruturas montadas para abrigar todos os sensores e equipamentos necessários (THRUN, 2010). Em muitos casos o custo desses equipamentos além do próprio carro em si torna proibitivo o estudo da área.

Nesse contexto surgem alternativas como o uso de mini-veículos como plataforma de estudo e teste de algoritmos de direção autônoma. Utilizando versões modificadas de carros de controle remoto ou construídos especificamente para este fim esses veículos geralmente usam sensores de baixo custo e *hardware* bem menos potente que seus equivalentes da vida real.

Com os recentes avanços no campo da visão computacional em relação à inteligência artificial, o aprendizado profundo vem gradativamente substituindo os métodos mais clássicos como fusão de sensores e outros métodos estatísticos.

Apesar do sucesso obtido nos trabalhos já citados, os veículos utilizados são equipados com dispositivos com um poder computacional alto quando comparados com o que é possível extrair de microcontroladores. Conseguir aplicar essas tecnologias em microcontroladores representaria um avanço substancial na área, visto que além do custo o consumo de energia seria drasticamente reduzido, o que é um fator de grande importância pois em sua maioria são alimentados por bateria.

Assim, surgem diversos trabalhos que tentam levar os avanços obtidos com o aprendizado profundo para plataformas equipadas com microcontroladores. Incluindo uma versão minificada de (BECHTEL et al., 2018) que substitui a *Raspberry Pi 3B+* por um microcontrolador *Raspberry Pi Pico*. Para fins de comparação, a Tabela 2.2 mostra a diferença entre o *hardware* de uma *Raspberry Pi 3B+* e o de uma *Raspberry Pi Pico*.

Peça	Raspberry Pi 3 Model B+	Raspberry Pi Pico
CPU	Broadcom BCM2837B0, Cortex-A53 (ARMv8) 64-bit SoC @ 1.4GHz	RP2040 2x Cortex-M0+@133MHz
Memória	1GB LPDDR2 SDRAM	264KB SRAM
Armazentamento	Cartão SD de até 32GB	$2MB$ de *flash*
Eneriga	$2,5A$	$< 100mA$

Tabela 2.2: Comparativo entre o hardware de uma *Raspberry Pi 3B+* e o de uma *Raspberry Pi Pico*.

[1] *Light Detection and Ranging*
[2] *Global Positioning System*

3

Metodologia

3.1 Simulação

A simulação do ambiente foi desenvolvida utilizando a linguagem de programação *Python*[3]. Ela consiste em um veículo representando o agente e uma cena 3D representando a área em que ele pode se locomover de acordo com a ação escolhida. A obtenção do estado é feita por meio de uma câmera que fica acoplada à parte frontal do veículo, simulando a visão do motorista. Por padrão, toda a renderização é feita *"off-screen"*, isto é, o programa não necessita de uma janela para executar. Porém é possível configurar o simulador para que ele também renderize a cena em uma janela utilizando um ponto de vista em 3ª pessoa. A Figura 6 mostra um exemplo de estado obtido pelo agente durante a execução da simulação. A Figura 7 mostra a visualização da simulação, contendo além da cena 3D informações sobre o veículo.

A cena 3D é composta por um plano que representa o terreno e nele é aplicada uma textura de grama, uma *skybox* representando o céu e a área contendo as ruas, casas e demais objetos. Os modelos 3D dos objetos e a *skybox* foram feitos utilizando o *software* de modelagem 3D *blender*[4], enquanto terreno, ruas, calçadas e casas são geradas proceduralmente a partir das

[3] *Python* 3.10.0 <https://www.python.org/>
[4] blender 4.2 <https://www.blender.org/>

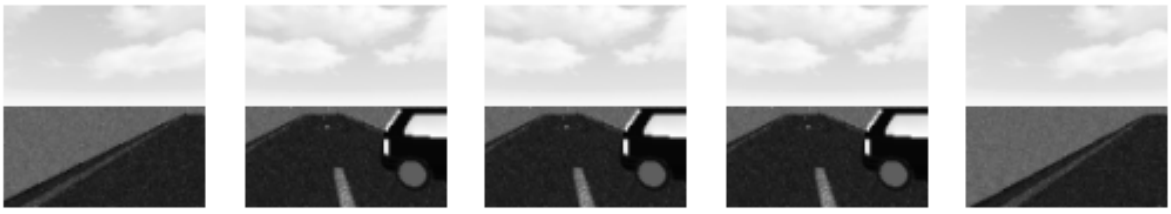

Figura 6: Exemplos de estado obtidos na simulação
Fonte: Autor.

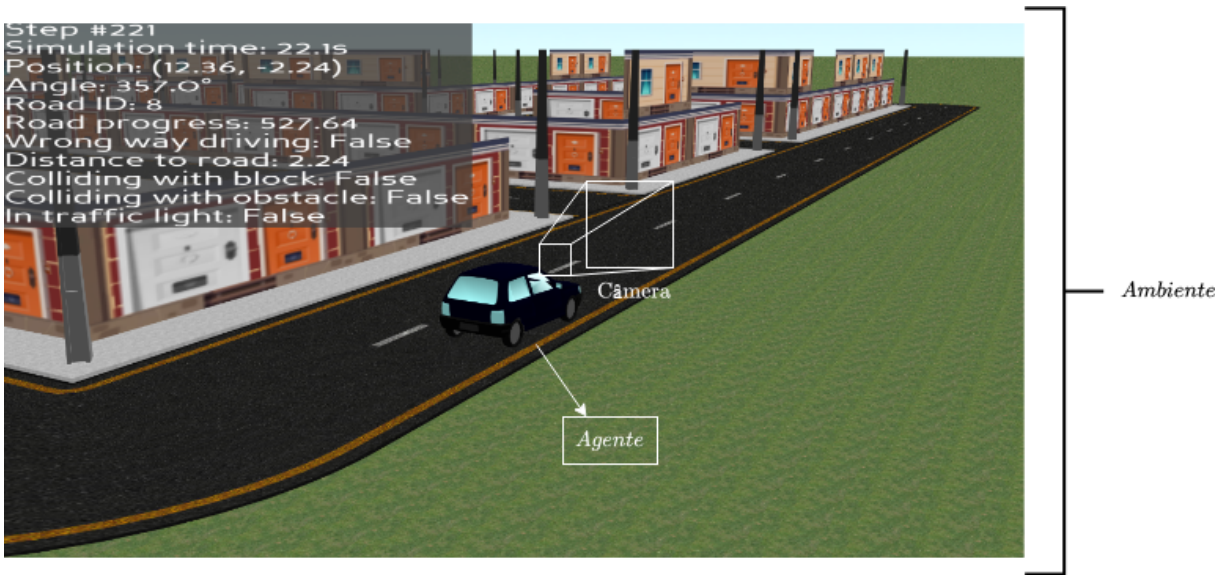

Figura 7: Captura de tela da simulação implementada, exibindo o ambiente, o agente e a câmera virtual presa ao veículo, que serve como o meio utilizado por ele para observar o estado do ambiente.

Fonte: Autor.

informações do mapa. Para a renderização foi utilizado o motor de jogos *Panda3D*[5] devido à possibilidade dele ser usado diretamente em *scripts Python*, o que facilitou a integração com o restante dos componentes da aplicação.

Ambiente *gym* customizado

O fluxo de execução do simulador seguiu as diretrizes da biblioteca *gymnasium*[6], que fornece uma *API* bastante consolidada no campo do aprendizado por reforço para a definição de ambientes. Isso permite que o simulador seja utilizado não somente pelos meios apresentados neste trabalho, mas por qualquer outro que seja compatível com a *API*. A partir da documentação da biblioteca foi criado um ambiente customizado para o problema proposto. Os métodos da interface implementados foram:

- **__init__**: Inicializa o ambiente utilizando os parâmetros utilizados na criação do ambiente. Também define variáveis importantes para a execução do ambiente, como o espaço de estados e o espaço de ações. Como o estado do problema se trata de uma imagem, foi utilizado um espaço do tipo *Box* com formato $(128, 128, 3)$ contendo os valores de cada *pixel* no intervalo $[0, 255]$. Já para o espaço de ações foi utilizado um do tipo *Discrete* onde cada ação é representada por um número de 0 a n, onde n é o número de ações possíveis.

- **reset**: Reinicia o ambiente e retorna o estado inicial e demais informações adicionais.

[5] *Panda3D* 1.10.14 <https://www.panda3d.org/>
[6] *gymnasium 0.29.1*<https://gymnasium.farama.org/>

O processo de escolha da posição e orientação do veículo é feito com o objetivo de dar variabilidade ao ambiente e será explicado posteriormente.

- **step**: Executa um passo da simulação a partir da ação recebida como parâmetro de acordo com o modelo cinemático do veículo e retorna o novo estado, a recompensa obtida neste passo, o estado de terminação do episódio e as informações adicionais do ambiente.
- **render**: Método utilizado para visualizar o ambiente. Caso o ambiente tenha sido criado com o parâmetro *render_mode* definido como *"human"* renderiza o ambiente do ponto de vista de 3ª pessoa em uma janela. É utilizado apenas para fins de visualização e demonstração.
- **close**: Libera os recursos utilizados pela simulação, como janelas criadas.

Geração das malhas 3D

Com a estrutura básica funcionando foi necessário modelar computacionalmente o mapa. Isto porque além da malha 3D seria preciso ter uma espécie de estrutura de dados capaz de fornecer informações relacionadas à pose do veículo em relação às ruas, por exemplo, para calcular as recompensas e decidir a validade da pose do veículo. Para isso, foi utilizada a biblioteca de geometria computacional *shapely*[7]. Com ela é possível criar várias entidades geométricas e realizar consultas espaciais nas mesmas. Assim, o modelo do mapa é composto por segmentos de reta concatenados representando as vias e polígonos que representam os quarteirões. A Figura 8 demonstra essa representação.

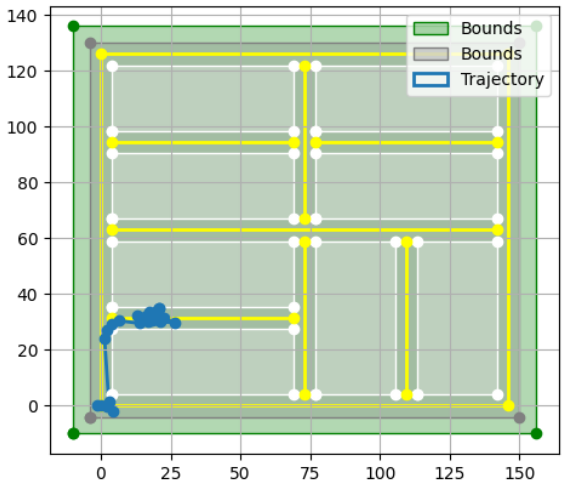

Figura 8: Representação geométrica do modelo do mapa
Fonte: Autor.

A malha 3D das ruas é obtida através da extrusão dos segmentos de reta que as representam, formando um quadrilátero. Para texturizar essa malha foi utilizada uma imagem de um trecho

[7] *shapely* 2.0.3 <https://shapely.readthedocs.io/en/2.0.3>

de asfalto contendo as faixas central e laterais. As coordenadas de textura horizontais levam em consideração de qual lado o vértice está em relação à linha de referência no sistema de coordenadas da via. Os vértices à esquerda recebem $u = 0$ enquanto os vértices à direita recebem $u = 1$. Já as verticais são calculadas levando em consideração o comprimento do segmento. A textura é configurada para repetir caso as coordenadas extrapolem. Dessa forma a textura é aplicada de forma contínua por todo o segmento. A Figura 9 exemplifica esse processo.

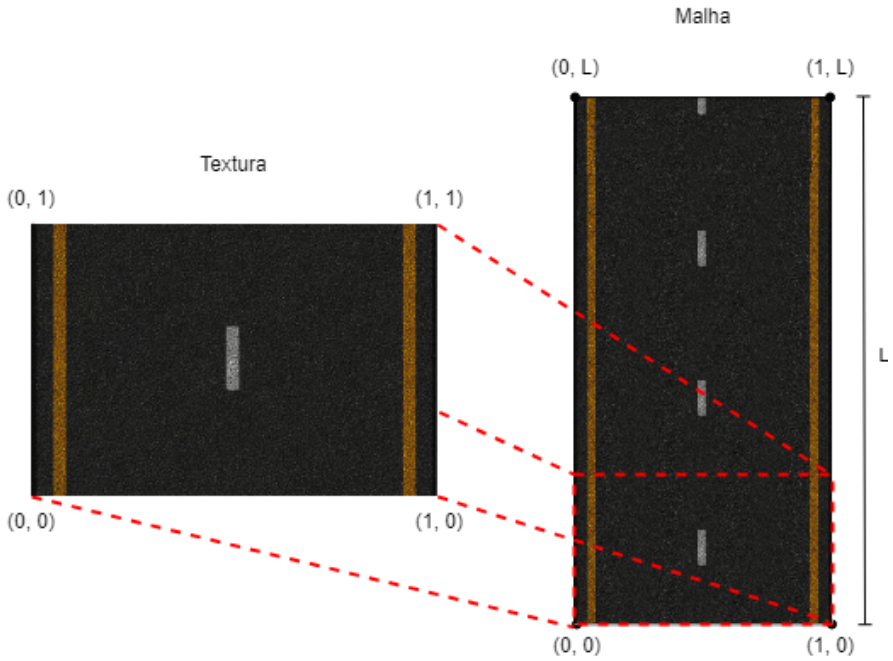

Figura 9: Mapeamento das coordenadas de textura de um segmento de rua de comprimento L.
Fonte: Autor.

Nos casos onde a linha de referência é composta por mais de um segmento é necessário combinar os quadriláteros de cada segmento sem que esses se sobreponham. Primeiramente é calculado para qual lado a curva se fecha. Em seguida é calculado o ponto de interseção das laterais do lado correspondente. A projeção desse ponto em cada segmento é usada para delimitar os quadriláteros gerados por cada um. Por fim, um segmento é ligado ao próximo por meio de um leque de triângulos iniciado no final do primeiro segmento e terminado no início do segmento seguinte do lado externo da curva com centro no ponto de interseção. Novamente as coordenadas de textura são mapeadas com o intuito de deixar a imagem contínua e seguindo a curva.

Espaço de ações

O espaço de ações do agente compreende um conjunto de 9 ações, associadas ao números de 0 a 8. A Tabela 3.1 mostra o significado de cada ação, onde as colunas correspondem ao efeito da ação no acelerador do veículo ϕ enquanto as linhas se referem à direção ψ. As ações são executadas discretamente durante um período de $\tau = 0,1s$. A partir do número da ação, são

	$\phi = -1$	$\phi = 0$	$\phi = 1$
$\psi = -1$	0	1	2
$\psi = 0$	3	4	5
$\psi = 1$	6	7	8

Tabela 3.1: Espaço de ações do agente

encontradas as

A Equação 12 descreve a cinemática utilizada para calcular a pose do veículo, onde $v = 5m/s$ é a velocidade linear e $45°/s$ a velocidade angular.

$$\begin{bmatrix} x_{k+1} \\ y_{k+1} \\ \theta_{k+1} \end{bmatrix} = \begin{bmatrix} x_k \\ y_k \\ \theta_k \end{bmatrix} + \begin{bmatrix} -\phi * v * \sin\theta_{k+1}\tau \\ \phi * v * \cos\theta_{k+1}\tau \\ \phi * \psi * \omega * \tau \end{bmatrix} \tag{12}$$

Cálculo da recompensa

A função de recompensa foi idealizada com o intuito de incentivar o agente a progredir no mapa, seguindo as regras de trânsito. O progresso considerado é o quanto o veículo se desloca paralelamente à linha de referência da pista.

A cada vez que o agente se desloca $2,5m$ em pelo menos 10 passos de simulação é verificada sua orientação em relação à pista. Caso esteja dentro da pista a recompensa é dada de acordo com o sentido percorrido, isto é, se o progresso foi positivo ele recebe o valor 1, e -1 caso contrário. A Figura 10 mostra um diagrama exemplificando a recompensa obtida pelo agente em dois casos diferentes.

Para calcular o progresso do veículo é feita a projeção de sua posição na linha de referência da rua mais próxima com o auxílio da estrutura de dados citada anteriormente. O sentido correto também é determinado utilizando o vetor $(c_x, c_y) - (c_{x_{proj}}, c_{y_{proj}})$. Como $(c_{x_{proj}}, c_{y_{proj}})$ é o ponto mais próximo da posição do veículo na linha de referência, esse vetor por sua vez é perpendicular à linha de referência. Dessa forma, fazendo a rotação de 90º obtém-se o vetor com o sentido correto em relação à posição do veículo. Se o produto interno entre o vetor da via e o vetor do veículo for positivo então ele está orientado no sentido correto.

No caso do veículo se encontrar fora da pista ou dirigindo na contramão, ele recebe uma recompensa com valor igual a $-0,1$ por passo de simulação. Caso fique mais de $7,5m$ longe da pista o episódio é finalizado precocemente e é retornado uma recompensa com valor igual a -10. Exceto as situações listadas, o valor da recompensa é igual a 0.

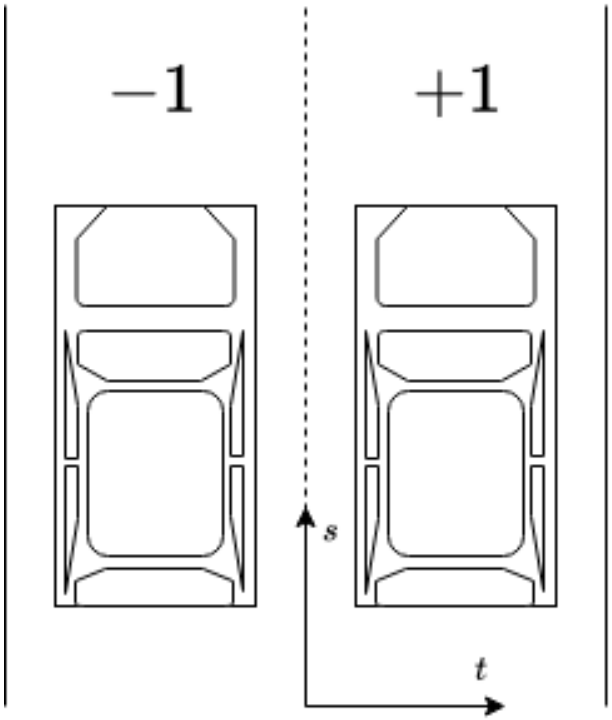

Figura 10: Diagrama demonstrando a recompensa obtida pelo agente. A linha tracejada representa a linha de referência da pista. No lado esquerdo, o veículo dirige na contramão e por isso em 10 passos receberá -1 no total. Já no lado direito, o veículo dirige na faixa correta e receberá 1

Fonte: Autor.

3.2 Treinamento e validação

O treinamento do agente foi feito através da implementação de *Deep Q-Learning* da biblioteca *Stable-Baselines 3*[8]. Foi utilizado um *replay buffer* de tamanho n_{buffer}. Cada sessão de treino teve a duração máxima de n_{steps} passos. A taxa de exploração inicia em ε_i e decai linearmente até ε_f durante os primeiros $n_{exploration}$ passos da sessão, se mantendo constante até o final do treinamento.

A rede neural utilizada pelo algoritmo é implementada com a biblioteca *PyTorch*[9]. Por padrão, a arquitetura segue a relatada em (MNIH et al., 2013). Para diminuir o tamanho do modelo, o extrator de características foi substituído por um que possui camadas de *pooling* entre as camadas convolucionais. A Tabela 3.2 mostra a arquitetura final utilizada no modelo. Ela foi gerada a partir do modelo no formato *ONNX* [10] utilizando sua *API Python*.

[8] *Stable-Baselines 3* 2.3.2 <https://stable-baselines3.readthedocs.io/>
[9] *PyTorch* 2.4.1 <https://pytorch.org/>
[10] *Open Neural Network Exchange* <https://onnx.ai>

Camada	Tamanho da entrada	Tamanho da saída	Parâmetros
Conv1	$1 \times 96 \times 96 \times 1$	$1 \times 94 \times 94 \times 16$	160
MaxPool1	$1 \times 94 \times 94 \times 16$	$1 \times 47 \times 47 \times 16$	0
Conv2	$1 \times 47 \times 47 \times 16$	$1 \times 45 \times 45 \times 16$	2.320
MaxPool2	$1 \times 45 \times 45 \times 16$	$1 \times 22 \times 22 \times 16$	0
Conv3	$1 \times 22 \times 22 \times 16$	$1 \times 20 \times 20 \times 16$	2.320
MaxPool3	$1 \times 20 \times 20 \times 16$	$1 \times 10 \times 10 \times 16$	0
Flatten	$1 \times 10 \times 10 \times 16$	1×1600	0
FullyConnected1	1×1600	1×128	204.928
FullyConnected2	1×128	1×64	8.256
FullyConnected3	1×64	1×64	4.160
FullyConnected4	1×64	1×9	585
Total			222.729

Tabela 3.2: Arquitetura da rede utilizada para treinar o agente

Avaliação do agente

A métrica utilizada para avaliar a performance do agente foi a média do retorno. Durante o treinamento, o modelo é submetido periodicamente a uma etapa de avaliação que registra a performance obtida. Essas informações são observadas e permitem adotar estratégias como salvar o modelo com o melhor resultado ou interromper o treinamento caso o agente consiga realizar a tarefa com sucesso. Para monitorar essas métricas foi utilizada a ferramenta *Tensorboard*[11]. Ela é alimentada com os dados coletados durante o treinamento automaticamente pelo algoritmo de treinamento. A Figura 11 mostra uma captura de tela da ferramenta em funcionamento.

Com o modelo treinado foram realizados diversos experimentos para verificar a capacidade de generalização do agente. Primeiramente ele foi testado em um ambiente semelhante ao utilizado no treinamento. Em seguida, com auxílio da modelagem de mapa feita para a simulação, foram criados cenários com situações inéditas para o modelo como pistas contendo curvas.

3.3 Implantação do modelo embarcado

Hardware

Os testes em ambiente real foram feitos por meio da combinação de um robô móvel e um módulo de visão computacional. Para o módulo de visão computacional foi escolhido o *kit* de desenvolvimento *T-Camera S3*[12] da *LILYGO* [13] exibido na Figura 12. Ele conta com uma câmera de 2MP (*megapixels*) e um chip da família *ESP32-S3*[14], que possui aceleração para tarefas envolvendo inteligência artificial.

[11] *Tensorboard* 2.8.0 <>https://www.tensorflow.org/tensorboard
[12] T-Camera S3 <https://lilygo.cc/products/t-camera-s3>
[13] LILYGO<https://lilygo.cc/>
[14] ESP32-S3 <https://www.espressif.com/en/products/socs/esp32-s3>

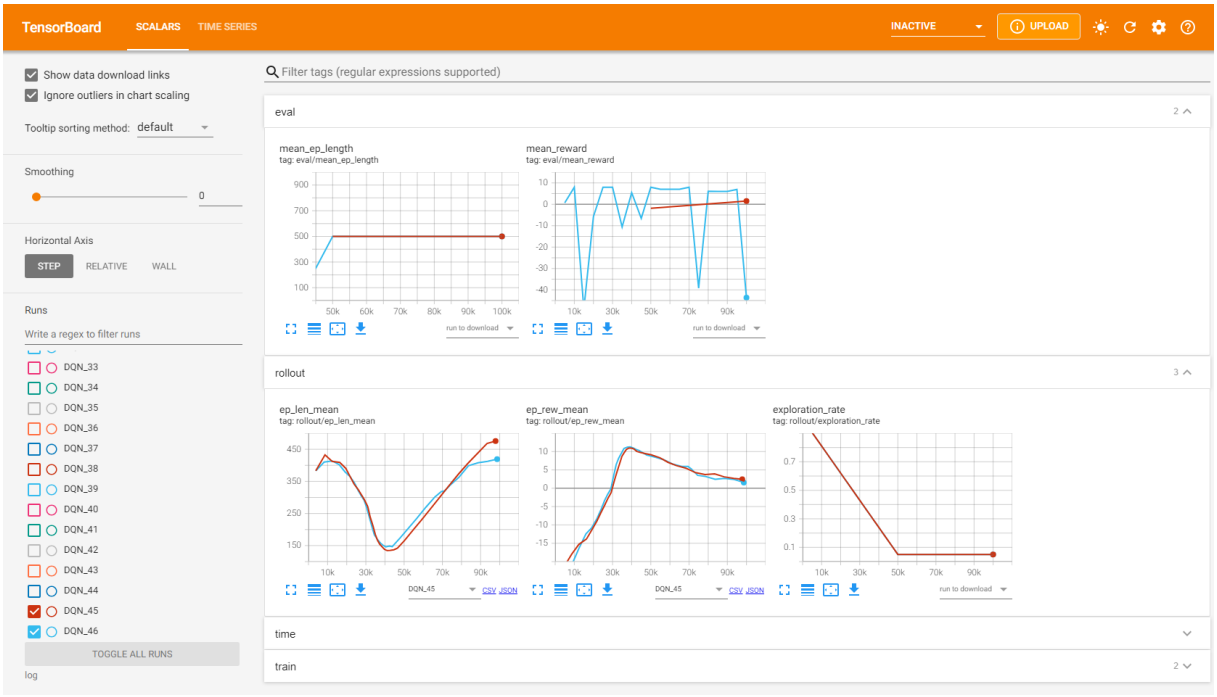

Figura 11: Captura de tela da ferramenta *Tensorboard*. Nela é possível visualizar as métricas capturadas durante os treinamentos em tempo real.
Fonte: Autor.

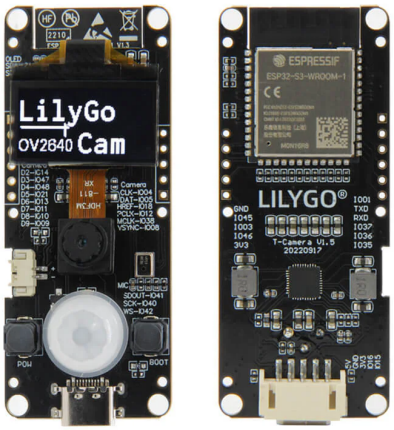

Figura 12: *Kit* de desenvolvimento *T-Camera S3* da *LILYGO*.

Conversão do modelo

Como dito anteriormente, a biblioteca utilizada para obter o modelo do agente utiliza o *PyTorch* como *back-end* para as redes neurais. Embora seja uma alternativa bastante conhecida pela comunidade científica, não seria possível usar diretamente os modelos criados com ele no

sistema embarcado, visto que seria utilizado o *Tensorflow Lite*[15] para este fim. Sendo assim, foi necessário realizar a conversão entre o modelo para torná-lo compatível com o objetivo deste trabalho.

Devido às diferenças nas implementações dos dois *frameworks*, apenas transferir os valores dos pesos de um modelo para outro equivalente é uma estratégia com um grande potencial de erros. Para assegurar que o modelo convertido tivesse o mesmo funcionamento do original optou-se pelo uso do *ONNX* como formato intermediário. Com ele foi possível converter o modelo original no formato do *PyTorch* para um modelo *Tensorflow*[16] através da biblioteca *onnx2tensorflow*[17]. A partir daí, foi utilizado módulo de conversão do próprio *Tensorflow* para converter o modelo para o formato do *Tensorflow Lite*. Esse processo resulta em um modelo que pode ser usado para realizar inferências em dispositivos com restrições de processamento e energia, como dispositivos móveis ou microcontroladores.

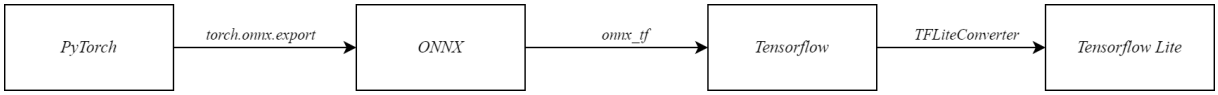

Figura 13: Etapas utilizadas na conversão do modelo *Pytorch* em um modelo do *Tensorflow Lite*
Fonte: Autor.

Além disso, foi necessário utilizar a ferramenta *ONNX Modifier*[18] para editar o modelo *ONNX* antes de convertê-lo para o formato *Tensorflow*. Por padrão, o modelo criado no treinamento utiliza uma camada com a operação *argmax* para obter a ação desejada. No entanto essa camada acaba utilizando um valor inteiro de 64 *bits* como valor de saída, o que não é suportado pelo ambiente de execução do *Tensorflow Lite Micro*. Sendo assim, o modelo final foi editado removendo a camada *argmax* e redirecionando para a saída da rede a penúltima camada que contém a distribuição de probabilidade das ações. A camada *argmax* era utilizada apenas por conveniência e sua remoção não teve nenhum impacto na performance do modelo pois não interferiu em nada na quantidade ou nos valores dos pesos. A única consequência disso foi a necessidade de implementar no *firmware* o processamento da distribuição para definir a ação escolhida.

Desenvolvimento de *firmware*

O *firmware* do módulo de visão computacional possui duas variantes, uma onde a inferência é feita em um servidor na nuvem e o microcontrolador se conecta a ele através de uma rede *Wi-Fi* e outra onde todo o processamento é feito no próprio microcontrolador utilizando um modelo embarcado. As duas versões foram feitas com o ambiente de programação ESP *IoT Development Framework* (IDF)[19] utilizando a mesma base de código, com as diferenças sendo

[15] *Tensorflow Lite* <https://www.tensorflow.org/lite/guide>
[16] *Tensorflow* <https://www.tensorflow.org>
[17] *onnx2tensorflow* 1.10.0 <https://github.com/onnx/onnx-tensorflow>
[18] *ONNX Modifier* <https://github.com/ZhangGe6/onnx-modifier>
[19] *ESP-IDF* <https://idf.espressif.com/>

controladas por meio de *flags* de compilação.

Na versão com conexão à *Internet*, após a inicialização, o sistema tenta se conectar à rede *Wi-Fi* com as credenciais pré configuradas. Após conseguir estabelecer a conexão a câmera é inicializada e a partir dela as imagens são capturadas e postas numa fila. Em seguida a imagem é formatada em uma *string json* contendo os valores dos *pixels* e enviada ao servidor do *Google Cloud Platform*[20] para inferência enquanto o sistema entra no modo de espera pelo resultado. Após obter a resposta do servidor com a inferência esse resultado é enviado via protocolo *UART* pela porta serial do dispositivo.

Já na versão de *TinyML* todo o processamento é feito pelo próprio microcontrolador utilizando a biblioteca *Tensorflow Lite Micro*. O modelo obtido após a etapa de conversão é embarcado diretamente no código como um *array* de *bytes*. Durante a inicialização esse *array* é usado para reconstruir o modelo dentro do *firmware*. Assim como na versão anterior, após inicializar a câmera as imagens são capturadas e enviadas para uma fila. No entanto, ao invés de codificar a imagem e enviar para um servidor nessa versão a imagem é usada diretamente como entrada para o modelo carregado. Dessa forma a inferência é obtida e enviada pela porta serial.

Avaliação do sistema

Para comparar as duas versões de *firmware* implementadas foi calculado o tempo total entre a captura da imagem e a obtenção da inferência. Na primeira versão esse tempo inclui a codificação e envio da imagem para o servidor, a inferência do modelo na nuvem e o tempo de recebimento da resposta. Já na segunda versão o tempo total corresponde ao tempo levado para realizar a inferência do modelo embarcado. O tempo total é calculado multiplicando a quantidade de ciclos do processador entre o início e o fim das operações.

Também foi realizado uma comparação entre o modelo original do *PyTorch* e o modelo final otimizado do *Tensorflow Lite*, a fim de quantificar a redução da acurácia decorrente do processo de otimização. Foram executados 1000 passos de simulação utilizando o modelo original para escolher as ações. A cada passo, também é feita a inferência utilizando a *API* do *Tensorflow Lite* com o modelo otimizado, salvando os valores em duas listas diferentes.

[20] *Google Cloud Platform* <https://cloud.google.com/>

4

Resultados

Neste capítulo serão apresentados os resultados obtidos durante o desenvolvimento deste trabalho. Na Seção 4.1 são apresentados os resultados do treinamento, como o retorno em função do tempo de treinamento e características dos modelos. A Seção 4.2 exibe os resultados obtidos implantando o modelo num *kit* de desenvolvimento, tanto com *TinyML* quanto com conexão à *Internet*. Vídeos de demonstração do simulador, treinamento e testes foram disponibilizados em [21].

4.1 Treinamento

As Figuras 14 e 15 mostram a evolução da recompensa total e a duração do episódio, respectivamente, a cada 10 episódios completos. Na Figura 14 é possível observar que no a partir de cerca de 75% do treinamento o retorno obtido fica em torno de 40 , enquanto que na Figura 15 o mesmo acontece com a duração dos episódio, mas em torno de 500, ou seja, ao final do treinamento o agente conseguia concluir o episódio obtendo um retorno próximo ao máximo possível dada a função de recompensa.

A Tabela 4.1 descreve o espaço ocupado pelos modelos em relação a armazenamento e a taxa de compressão obtida em relação ao modelo original. O modelo original corresponde ao modelo utilizado na fase de treinamento. Já o modelo convertido se refere ao modelo resultante da conversão do modelo original para o formato *Tensorflow Lite*. O modelo otimizado é o modelo obtido a partir da aplicação das técnicas de otimização no modelo convertido. No caso do modelo original, foi levado em consideração o tamanho do arquivo *zip* gerado ao salvar o agente, que contém todas as informações para que o modelo possa ser restaurado. Já no caso dos modelos convertido e otimizado o tamanho se refere à quantidade de bytes que o modelo ocupa.

[21] <https://drive.google.com/drive/folders/1G2sCZeC0BewipQ0kCFIEsjjmgcZTwyrU>

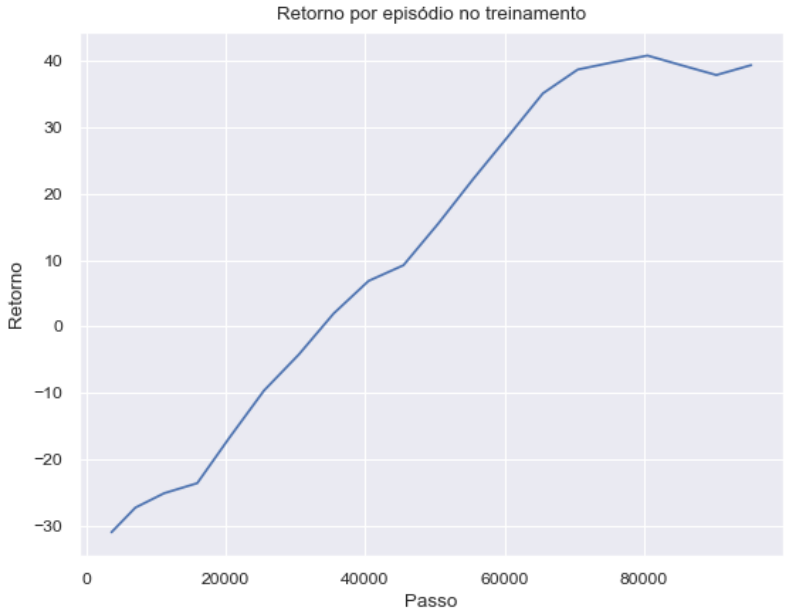

Figura 14: Retorno médio durante o treinamento.
Fonte: Autor.

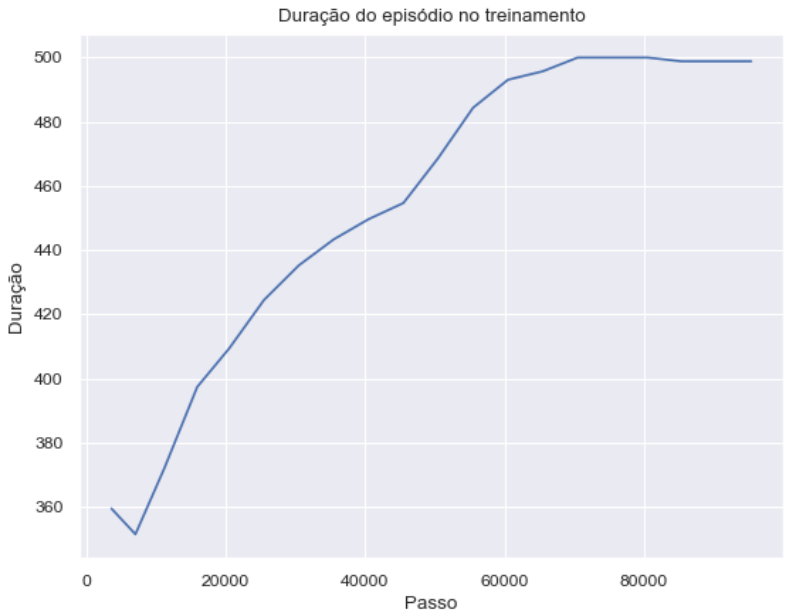

Figura 15: Duração média do episódio durante o treinamento.
Fonte: Autor.

Além disso, os experimentos feitos para quantificar a redução na acurácia após a otimização mostraram que, para as mesmas entradas, o modelo otimizado teve a mesma resposta que o modelo original em $88,4\%$ das vezes. Vale ressaltar que essa diferença não necessariamente reflete em perda de desempenho do modelo otimizado como num problema de classificação, mas apenas uma diferença nas ações tomadas. A Figura 16 compara o retorno obtido entre os

	Original	Convertido	Otimizado
Tamanho (*bytes*)	3.676.365	895.544	230.480
Taxa de compressão		75,64%	93,73%

Tabela 4.1: Espaço ocupado pelos modelos e taxa de compressão em relação ao original.

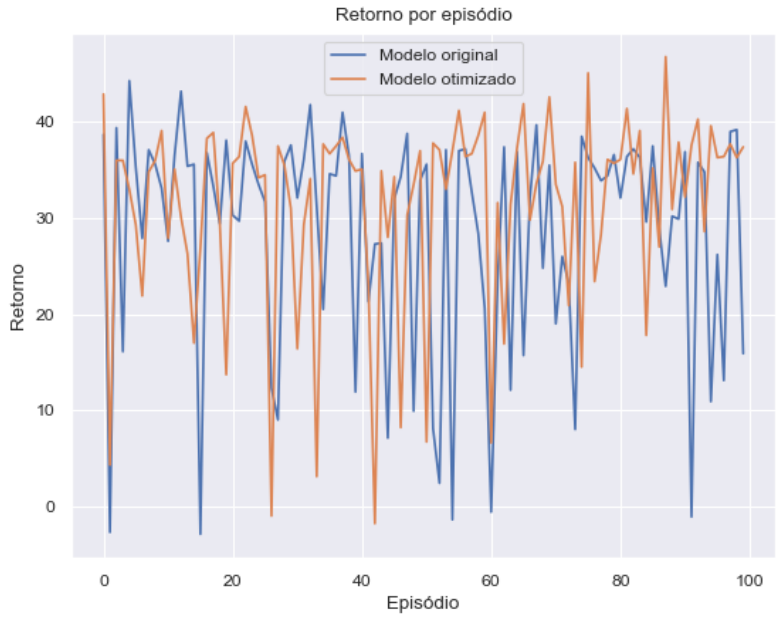

Figura 16: Comparação entre o retorno obtido com o modelo original e com o modelo otimizado em 100 episódios.

Fonte: Autor.

dois modelos, sendo executados em 100 episódios. A diferença entre o retorno médio obtido por cada modelo foi de aproximadamente 2,96.

A Figura 17 mostra o retorno médio obtido em cada momento de avaliação do modelo. O gráfico mostra o retorno médio obtido na execução de 30 episódios. Esse processo foi feito a cada 5.000 passos de simulação executados durante o treinamento.

4.2 Implantação

Com relação às duas abordagens utilizadas, a Tabela 4.2 mostra a latência obtida em cada uma delas. A coluna ”Embarcado” se refere à inferência feita no próprio dispositivo embarcado, sem a necessidade de conexão com a *internet*. A coluna ”Nuvem” se refere à inferência feita em um servidor, onde existe a necessidade do dispositivo se conectar à *internet* para enviar a imagem de entrada e receber a ação como resposta. Na Figura 18 é possível ver o tempo levado para cada inferência em cada abordagem.

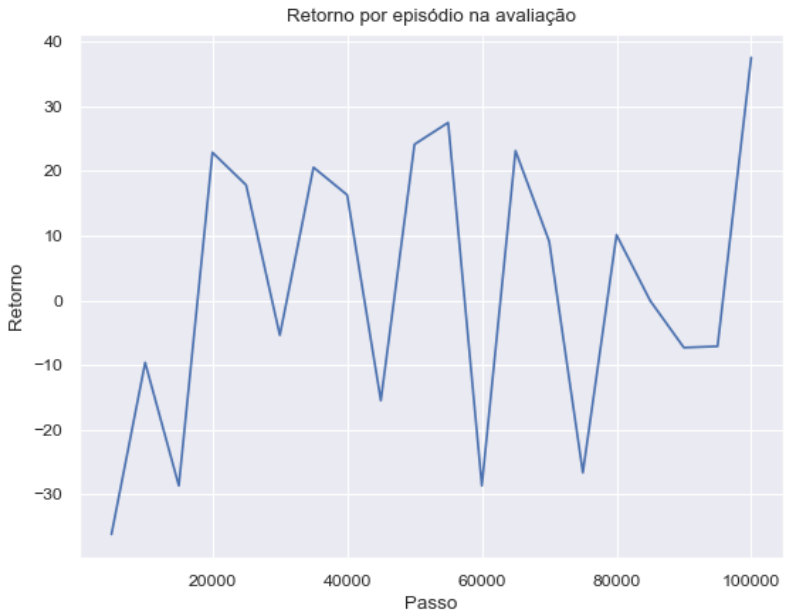

Figura 17: Retorno obtido pelo agente na avaliação durante o treinamento.
Fonte: Autor.

	Tempo de inferência (ms)			
	Mínimo	Máximo	Médio	Desvio padrão
Embarcado	200	200	200	0
Nuvem	480	3320	563	200

Tabela 4.2: Tempo de inferência das duas abordagens de implantação do modelo.

4.3 Síntese dos Resultados

Pela natureza da função de recompensa utilizada, onde o agente só consegue aumentar o retorno caso se locomova para frente, a estabilidade ao final do treinamento indica que o agente aprendeu a realizar a tarefa de dirigir corretamente, sem finalizar os episódios precocemente por sair da pista ou dirigir na contra-mão recebendo valores de recompensa negativos.

O retorno médio obtido na execução dos episódios de avaliação do agente acompanha a performance apresentada no treinamento, indicando que o agente não apenas foi capaz de aprender a solucionar o problema como também foi capaz de generalizar a solução para uma situação com maior nível de dificuldade.

Com relação à conversão do modelo, a redução significativa do tamanho do modelo convertido para o modelo otimizado se dá por meio da quantização dos parâmetros, que permite armazenar os valores como inteiros de 8 *bits* ao invés de números de ponto flutuante de 32 *bits*. Essa redução facilita a implantação do modelo em um microcontrolador, que tem uma quantidade bastante limitada de memória.

Em relação à acurácia, apesar do modelo otimizado não obter as mesmas saídas que o mo-

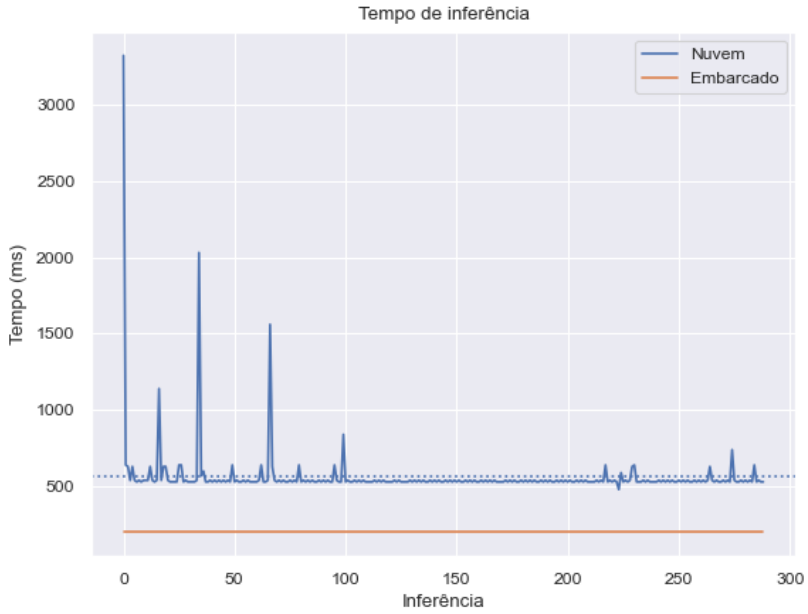

Figura 18: Tempo de inferência ao decorrer de um episódio com 100 passos.
Fonte: Autor.

delo original em alguns casos, no geral seu desempenho se mostrou satisfatório, alcançando valores semelhantes de retorno.

Já em relação à latência, enquanto no modelo embarcado o tempo de inferência permanece constante, no modelo na nuvem ele apresenta variações, com picos chegando a segundos. Pelo fato da conexão *HTTP* ser reutilizada durante as requisições, a primeira conexão apresenta um tempo de resposta consideravelmente maior que as outras. No entanto, ainda é possível observar a inconsistência que torna a alternativa não adequada para sistemas com requisitos de tempo real. Isso demonstra a importância que o *TinyML* representa para o desenvolvimento de sistemas embarcados que usam modelos de inteligência artificial, pois numa situação em que o acesso à *Internet* fosse ainda mais restrito esse tempo aumentaria muito mais, visto que o tráfego de informações entre o dispositivo e o servidor representa o maior gargalo da aplicação.

5

Conclusão

O desenvolvimento deste trabalho mostrou o fluxo de trabalho de utilização de redes neurais em microcontroladores, com aplicação na área de direção autônoma. Foram implementados a simulação do ambiente, o treinamento, a conversão e avaliação do modelo obitdo e a implementação de *firmware* capaz de executar o modelo em um *kit* de desenvolvimento, comparando com a execução do mesmo modelo na nuvem. O modelo conseguiu aprender a executar a tarefa de forma satisfatória e a abordagem do modelo embarcado superou a do modelo na nuvem no quesito de latência.

A Seção 5.1 apresenta os pricipais desafios e soluções encontrados. A Seção 5.2 elabora sobre as contribuições deixadas. A Seção 5.3 apresenta possíveis pontos de aprimoramento deste trabalho para trabalhos futuros.

5.1 Desafios e Soluções

Dentre os desafios encontrados na execução deste trabalho é possível destacar:

- **Elaboração da função de recompensa**: Durante os testes iniciais do treinamento o agente conseguia encontrar brechas na função de recompensa, fazendo com que ele conseguisse obter um resultado considerável quando visto apenas pelo ponto de vista numérico. No entanto, ao executar a política obtida no ambiente era possível notar que o comportamento apresentado não era o desejado. Para contornar esse problema a função de recompensa foi simplificada para eliminar estas possibilidades.

- **Incompatibilidade na conversão do modelo**: Devido ao modo em que os modelos são estruturados entre as diferentes bibliotecas utilizadas foi necessário dividir a conversão em algumas etapas. Além disso, devido à incompatibilidades entre os modelos gerados foi necessário realizar etapas manuais na conversão, como remoção de camadas.

5.2 Contribuições do Projeto

As principais contribuições feitas no desenvolvimento deste trabalho foram:

- **Ambiente**: O ambiente implementado pode ser utilizado para realizar experimentos, como também pode ser utilizado para treinar outros agentes com objetivos diferentes. Além disso, sua aplicação não se limita ao aprendizado por reforço, pois também pode ser utilizado para criar bases de dados para alimentar algoritmos de aprendizado supervisionado.
- **Fluxo de trabalho com *TinyML* e aprendizado por reforço**: O fluxo de trabalho seguido é independente do ambiente utilizado, ou seja, pode ser replicado para diferentes casos de uso.

5.3 Trabalhos Futuros

Com base no que foi apresentado, é possível elencar algumas alternativas para extender este trabalho, como:

- **Aprimorar a simulação**: Adicionar situações à simulação, como semáforos, outros veículos, obstáculos, condições climáticas etc.
- **Controle contínuo**: Utilizar um espaço de ações contínuo.
- **Adicionar informações ao estado**: Além da imagem, utilizar como entrada informações de sensores, como odometria nas rodas ou sensores de distância.
- **Experimentar outros algoritmos**: Tentar aplicar outros tipos de algoritmos de aprendizado por reforço, como *Actor-Critic* (AC) ou realizar transferência de conhecimento de um modelo já treinado.

Referências Bibliográficas

ABADADE, Y. et al. A comprehensive survey on tinyml. *IEEE Access*, IEEE, 2023.

BECHTEL, M.; WENG, Q.; YUN, H. Deeppicarmicro: applying tinyml to autonomous cyber physical systems. In: IEEE. *2022 IEEE 28th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. [S.l.], 2022. p. 120–127.

BECHTEL, M. G. et al. Deeppicar: A low-cost deep neural network-based autonomous car. In: IEEE. *2018 IEEE 24th international conference on embedded and real-time computing systems and applications (RTCSA)*. [S.l.], 2018. p. 11–21.

GRIGORESCU, S. et al. A survey of deep learning techniques for autonomous driving. *Journal of field robotics*, Wiley Online Library, v. 37, n. 3, p. 362–386, 2020.

HAN, L.; XIAO, Z.; LI, Z. Dtmm: Deploying tinyml models on extremely weak iot devices with pruning. *arXiv preprint arXiv:2401.09068*, 2024.

HUBEL, D. H.; WIESEL, T. N. et al. Receptive fields of single neurones in the cat's striate cortex. *J physiol*, v. 148, n. 3, p. 574–591, 1959.

HUFFMAN, D. A. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, IEEE, v. 40, n. 9, p. 1098–1101, 1952.

INTERNATIONAL, S. *Automated driving: Levels of driving automation are defined in new SAE international standard J3016*. [S.l.]: SAE International Warrendale, PA, USA, 2014.

LECUN, Y.; BENGIO, Y.; HINTON, G. Deep learning. *nature*, Nature Publishing Group UK London, v. 521, n. 7553, p. 436–444, 2015.

MNIH, V. et al. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. Disponível em: <http://arxiv.org/abs/1312.5602>.

NOVAC, P.-E. et al. Quantization and deployment of deep neural networks on microcontrollers. *Sensors*, MDPI, v. 21, n. 9, p. 2984, 2021.

ROSENBLATT, F. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, American Psychological Association, v. 65, n. 6, p. 386, 1958.

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

SWAMINATHAN, S. et al. Sparse low rank factorization for deep neural network compression. *Neurocomputing*, Elsevier, v. 398, p. 185–196, 2020.

THRUN, S. Toward robotic cars. *Communications of the ACM*, ACM New York, NY, USA, v. 53, n. 4, p. 99–106, 2010.

WATKINS, C. J. C. H. Learning from delayed rewards. King's College, Cambridge United Kingdom, 1989.