Coin Counting With Raspberry Pi Zero 2W

Rodrigo - 2022002936
Ryan - 2021020958
Roldão Neto - 2022002954

1. Definição do Problema e Motivação:

Definição do Problema: O projeto aborda o desenvolvimento de um sistema de visão computacional embarcado para a detecção, classificação e contagem em tempo real de três tipos específicos de moedas brasileiras (1 Real, 50 centavos e 10 centavos). O desafio central reside na severa restrição de hardware: a solução deve operar em um Raspberry Pi Zero 2W, um dispositivo com capacidade de processamento e memória extremamente limitadas, exigindo uma performance de inferência mínima de 0.5 FPS (Frames Per Second).

Motivação: A principal motivação do projeto é validar a viabilidade de arquiteturas modernas de detecção de objetos ultraleves (como MobileNetV2 0.35 + FOMO) para aplicações práticas de *TinyML* (Machine Learning em dispositivos de borda). O projeto serve como uma Prova de Conceito (POC) para demonstrar que é possível executar tarefas de visão computacional, como a distinção de objetos visualmente similares (moedas), em hardware de baixíssimo custo, utilizando técnicas agressivas de otimização (quantização INT8) e um processo rigoroso de curadoria de dados para superar as limitações do dispositivo.

2. Trabalhos Relacionados e Revisão da Literatura

A literatura sobre Visão Computacional Embarcada (*TinyML*) foca em superar o gargalo entre a alta demanda computacional dos modelos de *deep learning* e a severa restrição de *hardware* (CPU, RAM, energia) de dispositivos de borda. Esta revisão aborda quatro pilares que fundamentam este projeto: arquiteturas de *backbone* eficientes, *detection heads* ultraleves, técnicas de otimização de modelo e a mitigação de desafios de *dataset*.

2.1. Arquiteturas Leves para Dispositivos de Borda

A viabilidade de rodar CNNs em *hardware* limitado depende de arquiteturas eficientes. A família **MobileNet** tornou-se um padrão da indústria para essa finalidade. O **MobileNetV2** [1], especificamente, introduziu os conceitos de *inverted residuals* e *linear bottlenecks*, permitindo construir redes profundas com um custo computacional drasticamente reduzido. A variante `alpha=0.35`, escolhida para este projeto, reduz ainda mais a largura da rede (número de canais), sendo uma escolha estratégica para dispositivos com CPUs de baixo consumo, como o Raspberry Pi Zero 2W. Trabalhos como o de Chiu et al. (2020) [2] validam o uso do MobileNetV2 em sistemas embarcados, demonstrando seu equilíbrio entre acurácia e eficiência.

2.2. Detecção de Objetos em Tempo Real: De YOLO a FOMO

Enquanto modelos como o YOLO (You Only Look Once) são populares para detecção em tempo real, suas variantes, mesmo as "tiny", frequentemente exigem *hardware* mais robusto ou aceleradores dedicados (como TPUs) para atingir performance adequada [3]. Isso os torna inviáveis para o *hardware-alvo* deste projeto.

Como alternativa, a arquitetura **FOMO (Faster Objects, More Objects)** [4] da Edge Impulse representa uma abordagem mais alinhada ao *TinyML*. O FOMO reformula a detecção: em vez de prever caixas delimitadoras (*bounding boxes*), ele trata a tarefa como uma classificação de centroides em uma grade. Essa simplificação reduz drasticamente a complexidade do *detection head* e o pós-processamento, consumindo, segundo seus desenvolvedores, até 30 vezes menos recursos que o YOLOv5 [4]. Estudos recentes, como o de Lin et al. (2024) [5], validam a combinação de MobileNetV2 e FOMO em *hardware* embarcado, alcançando alta velocidade (55 FPS) e precisão, o que corrobora a escolha arquitetônica deste projeto.

2.3. Otimização de Modelo: Quantização INT8 e TFLite

A implantação em dispositivos de borda exige a compressão do modelo treinado. A **Quantização Pós-Treinamento (Post-Training Quantization - PTQ)** é a técnica padrão para isso. Conforme demonstrado na literatura [6, 7], a conversão dos pesos e ativações de ponto flutuante (`float32`) para inteiros de 8 bits (`INT8`) oferece um duplo benefício:

1. **Redução de Tamanho:** Reduz o tamanho do modelo em até 4x, vital para a RAM e armazenamento limitados.
2. **Aceleração de Inferência:** Operações com inteiros são computacionalmente mais eficientes em CPUs de baixo consumo, levando a ganhos de velocidade de 2x a 4x [7].

O *runtime* do **TensorFlow Lite (TFLite)** [8] é o framework padrão para executar esses modelos quantizados, fornecendo apenas o interpretador necessário para a inferência e minimizando a sobrecarga de *software* no dispositivo.

2.4. Desafios de Dados: *Domain Gap* e Pré-processamento

A literatura aponta dois desafios práticos comuns que foram centrais neste projeto:

- ***Domain Gap***: Trabalhos como o de Hou et al. (2021) [9] analisam o "domain gap" (lacuna de domínio), onde um modelo treinado em um conjunto de dados (domínio fonte, ex: datasets públicos) falha ao ser aplicado em condições do mundo real (domínio alvo, ex: câmera do Pi). A diferença em iluminação, ângulos de câmera e ruído invalida o modelo. A solução encontrada na literatura, e aplicada neste projeto, é a criação de um *dataset* customizado e controlado, capturado no próprio *hardware-alvo*.
- ***Data Augmentation***: Para robustecer *datasets* customizados (que são inerentemente pequenos), técnicas de *data augmentation* (rotação, ruído, zoom) são essenciais [10]. Plataformas como a Roboflow [11] são ferramentas comuns para aplicar sistematicamente essas transformações.

- **Consistência de Pré-processamento**: Um ponto crítico, muitas vezes subestimado, é a necessidade de manter consistência absoluta no pré-processamento entre o treinamento e a inferência [12, 13]. Como este projeto descobriu empiricamente, se o modelo foi treinado com entradas de 240x240 pixels, ele falhará se a inferência ao vivo fornecer uma resolução diferente (ex: 640x480). A literatura confirma que esta discrepância é uma fonte comum de falha de implantação.

Finalmente, embora a detecção de moedas seja um problema conhecido [14], trabalhos focados especificamente em moedas brasileiras [15] são menos comuns, e sua implementação em *hardware* tão restrito quanto o Pi Zero 2W contribui como uma validação prática relevante da arquitetura FOMO + MobileNetV2 para aplicações de *TinyML*.

3. Descrição do Dataset e Processo de Preparação:

- Dataset composto por fotos de moedas de 1 real, 50 e 10 centavos.
- Foi utilizado um fundo uniforme branco como base
- Um apoio para a câmera para melhor qualidade
- Movimentos nas moedas e variação na quantidade foram aplicados para uma melhor generalização do modelo.
- Após isso passamos por uma etapa de labeling usando roboflow com prompt assistant que consiste em um modelo pre treinado do roboflow que aprende seu padrão de labeling e automaticamente passa a realizar um predict das proximas labels acelerando esse processo que costuma ser demorado.
- Aplicamos ao final algumas técnicas de data augmentation como crop, flip, ajuste de brilho e contraste para gerar maior variedade de dados.
- E por fim o dataset foi exportado no formato

4. Arquitetura do Sistema e Diagrama:

A arquitetura do sistema foi projetada como uma aplicação web monolítica executada inteiramente no Raspberry Pi Zero 2W. Ela opera em um modelo cliente-servidor, onde o Pi Zero atua como o servidor (backend) e qualquer dispositivo na mesma rede (como um notebook ou smartphone) pode atuar como cliente (frontend) acessando a interface via navegador.

O sistema é dividido em três componentes principais:

1) Backend (Servidor Flask - `app.py`): O núcleo da aplicação, escrito em Python. É responsável por:

- **Gerenciamento da Câmera:** Utiliza a biblioteca `picamera2` em uma thread dedicada (`get_frame`) para capturar continuamente imagens da câmera e armazená-las na memória (`frame`), protegido por um `frame_lock`. Isso

desacopla a captura da câmera do processamento, garantindo que a inferência não bloqueie a aquisição de imagens.

- **Inferência de ML:** Carrega o modelo FOMO (`.lite`) usando `tflite_runtime`.
- **Servidor Web:** Utiliza o Flask para criar três rotas (endpoints):
  - `/`: Serve a página principal `index.html`.
  - `/video_feed`: Rota de streaming (MJPEG) gerenciada pela função `gen_frames`. Esta rota busca o frame mais recente, executa todo o pipeline de visão (pré-processamento, inferência `processar_frame`, pós-processamento com `cv2.connectedComponentsWithStats` e desenho de elipses) e transmite o resultado como um vídeo.
  - `/data`: Uma API JSON que fornece os resultados da contagem.
- **Gerenciamento de Estado (Thread-Safe):** A contagem de moedas (`counts`) e o valor total (`total`) são armazenados em uma variável global `global_data`. Para prevenir *race conditions* (conflitos de leitura/escrita), um `threading.Lock` (`data_lock`) é usado. A rota `/video_feed` (escritora) e a rota `/data` (leitora) só podem acessar essa variável após adquirir o *lock*.

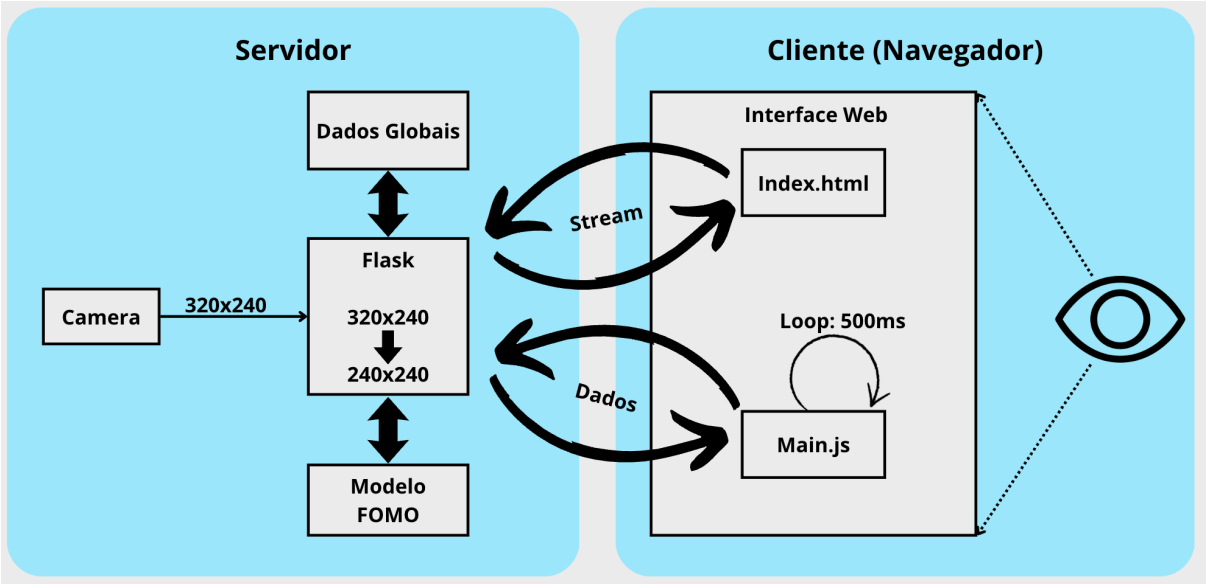
2) Frontend (Cliente Web - `index.html`, `main.js`, `styles.css`):

- `index.html`: Renderizado pelo Flask, estrutura a página. Contém uma tag `<img>` que aponta para a rota `/video_feed`, exibindo o stream ao vivo.
- `main.js`: Contém a lógica do cliente. Ele executa uma função `atualizarValores` a cada 500ms (`setInterval`). Esta função usa a API `fetch` para fazer uma requisição à rota `/data`, receber o JSON com as contagens e atualizar o texto (DOM) da página.

3) Modelo de ML (`.lite`): O artefato treinado e quantizado (conforme Seção 8) que é carregado pelo `tflite_runtime` no backend para realizar a detecção dos centroides das moedas.

O fluxo de dados principal é assíncrono: o frontend busca o stream de vídeo (`/video_feed`) e, independentemente, busca os dados de contagem (`/data`) em intervalos curtos, proporcionando uma experiência de usuário fluida onde os números atualizam quase em tempo real sem recarregar a página.

Diagrama de Blocos da Arquitetura:



5. Hardware Setup:

3. Configuração de Hardware (Hardware Setup)

A plataforma de hardware central para este projeto, definida pelas restrições de TinyML, foi o Raspberry Pi Zero 2W. Este dispositivo foi escolhido por seus recursos computacionais severamente limitados, que representam o desafio central do projeto:

- CPU: 1GHz quad-core 64-bit Arm Cortex-A53
- RAM: 512MB SDRAM

3.1. Sistema Operacional e Acesso

O dispositivo foi configurado com o Raspberry Pi OS (baseado em Debian Bookworm), a versão mais recente e otimizada para a arquitetura ARM do Pi. A instalação foi realizada utilizando a ferramenta Raspberry Pi Imager.

Para a operação, foi adotada uma abordagem "headless" (sem monitor ou interface gráfica direta). Todo o desenvolvimento, configuração e execução de testes foram gerenciados remotamente a partir de um computador host via:

- SSH (Secure Shell): Para acesso ao terminal e execução de comandos.
- SCP (Secure Copy Protocol): Para a transferência de scripts de inferência e do modelo .tflite treinado.

3.2. Periféricos e Otimizações de SO

- Câmera: Um módulo de câmera foi conectado à porta CSI (Camera Serial Interface) para a captura de imagens. A biblioteca libcamera foi utilizada

como a interface de software para acessar o stream de vídeo e capturar os frames para a inferência.

- Otimização de Memória (SWAP): Dada a RAM física extremamente limitada de 512MB, a memória SWAP do sistema foi aumentada. O valor padrão (100MB) foi expandido para 2GB (2000MB) editando o arquivo /etc/dphys-swapfile. Esta etapa foi crucial para evitar que o sistema operacional encerrasse o processo de inferência por falta de memória ao carregar o modelo e suas dependências.

6. Dependências de Software e Guia de Instalação:

Esta seção detalha o software necessário e os passos para configurar o ambiente de execução no Raspberry Pi Zero 2W, utilizando o arquivo de dependências fornecido.

6.1. Guia de Instalação:

O projeto requer o seguinte ecossistema de software:

- **Sistema Operacional:** Raspberry Pi OS (Bullseye ou superior, que utiliza a stack `libcamera` necessária para a `picamera2`).
- **Linguagem:** Python 3.x.
- **Bibliotecas Python (Principais):** As dependências exatas estão listadas no arquivo `requirements-rasp.txt`. As bibliotecas chave para a execução do `app.py` incluem:
  - `Flask==3.1.2`
  - `picamera2==0.3.30`
  - `tflite-runtime==2.14.0`
  - `numpy==1.24.2`
  - `opencv-python==4.12.0.88`

6.2. Guia de Instalação:

1) Clone o repositório do projeto para o dispositivo:

```
git clone https://github.com/Roldao-Neto/Coin-Counting-FOMO.git
```

2) Instalação das Dependências via requirements-rasp.txt: Recomenda-se o uso de um ambiente virtual (`venv`) para isolar as dependências do projeto.

sudo apt install python3-pip python3-venv

python3 -m venv venv

source venv/bin/activate

pip install -r requirements-rasp.txt

3) Estrutura de Arquivos: Certifique-se de que a estrutura de arquivos (conforme `README.md`) está correta e que o arquivo do modelo TFLite (ex: `ei-contador_moedas_final-lite-float.lite`) está presente no diretório raiz, ou ajuste a variável `model_path` em `app.py`.

4) Execução da Aplicação: Com o ambiente virtual ativado, inicie o servidor Flask:

python app.py

O servidor estará acessível na rede local através do IP do Raspberry Pi na porta 5000 (ex: `http://[IP_DO_PI]:5000`).

7. Arquitetura do Modelo e Metodologia de Treinamento:

A seleção da arquitetura e da metodologia foi o primeiro passo crítico para garantir a viabilidade do projeto, dada a severa restrição de rodar em tempo real (inferência ≥0.5 FPS) em um **Raspberry Pi Zero 2W**.

7.1. Arquitetura do Modelo:

Optou-se por uma arquitetura comprovadamente leve e eficiente, composta por um *backbone* **MobileNetV2 (alpha=0.35)** e um *detection head* **FOMO (Faster Objects, More Objects)**.

1. **Backbone (MobileNetV2 0.35):** O MobileNetV2 é uma arquitetura CNN padrão para dispositivos de borda. A variante `alpha=0.35` foi especificamente escolhida por reduzir drasticamente a largura da rede (número de canais). Esta decisão minimiza a carga computacional, sendo essencial para a CPU limitada do Pi Zero 2W.
2. **Detection Head (FOMO):** A escolha do FOMO foi uma decisão estratégica. Tendo em vista experiências anteriores com modelos mais pesados (como o YOLO), que se mostraram lentos no hardware-alvo, o FOMO ofereceu uma alternativa muito mais leve. Ele trata a detecção como uma classificação de centroides em uma grade, uma abordagem computacionalmente mais simples que se alinhava perfeitamente com os requisitos de velocidade do projeto.
3. **Tamanho da Entrada (Input Size):** O modelo foi configurado para uma entrada de **240×240 pixels**. Esta resolução foi selecionada como um balanço ideal entre acurácia (permitindo ao modelo "ver" detalhes suficientes nas moedas) e performance (mantendo a carga de processamento baixa). Como foi descoberto posteriormente (detalhado na Seção 7), **manter essa resolução de forma consistente** entre o treinamento e a inferência provou ser um dos fatores mais críticos para o sucesso da detecção no mundo real.

7.2. Metodologia de Treinamento:

O processo de treinamento foi gerenciado pela plataforma **Edge Impulse**, utilizando *transfer learning*.

1. **Estratégia de Treinamento (Transfer Learning):** O *backbone* MobileNetV2 foi inicializado com pesos pré-treinados no ImageNet. O treinamento (fine-tuning) focou em ajustar esses pesos e treinar a nova cabeça de detecção FOMO especificamente para a difícil tarefa de distinguir moedas, que são objetos visualmente muito similares entre si.
2. **Divisão do Dataset:** O conjunto de dados customizado (cuja criação é detalhada na Seção 7) foi dividido em **80% para treinamento** e **20% para validação**.
3. **Hiperparâmetros de Treinamento:** Os parâmetros chave para o treinamento foram:
    - **Otimizador:** `Adam`
    - **Taxa de Aprendizado (Learning Rate) Inicial:** `0.001`
    - **Épocas (Epochs):** `60` (máximo)
    - **Tamanho do Lote (Batch Size):** `32`
    - **Função de Perda (Loss Function):** Uma função de **Cross-Entropy Ponderada** (`weighted_xent`) foi utilizada, com um `object_weight=100`. Este peso foi vital: ele força o modelo a dar 100 vezes mais importância a uma célula contendo uma moeda do que a uma célula de fundo. Isso combate o desbalanceamento natural (muito mais fundo do que moedas) e foi crucial para que o modelo aprendesse a detectar os objetos de interesse.
4. **Callbacks e Otimização do Treinamento:** O treinamento foi supervisionado por *callbacks* do Keras para garantir a seleção do melhor modelo possível:

    - **`ReduceLROnPlateau` e `EarlyStopping`:** Foram usados para monitorar a perda de validação (`val_loss`), ajustando a taxa de aprendizado e interrompendo o treinamento se não houvesse melhoria, prevenindo *overfitting*.
    - **`ModelCheckpoint` (Foco no F1-Score):** Mais importante, o modelo foi configurado para salvar *apenas* a versão que alcançasse o **maior F1-Score de validação** (`val_f1`). A escolha do F1-Score (em vez da *loss*) como métrica de seleção foi deliberada, pois ela mede diretamente a qualidade da *detecção* (equilíbrio entre precisão e recall), que era o objetivo final do projeto, e não apenas a acurácia da classificação.

- Optimization techniques for edge deployment - Rodrigo

8. Técnicas de Otimização para Implantação em Borda (Edge)

A estratégia de otimização foi uma continuação direta das escolhas de arquitetura feitas na Seção 4. Tendo já selecionado um *backbone* (MobileNetV2 0.35) e *head* (FOMO) inerentemente leves, a otimização final focou em converter este modelo para extrair a performance máxima da CPU do Raspberry Pi Zero 2W.

8.1. Quantização Pós-Treinamento (INT8)

A principal técnica de otimização de *deploy* foi a **Quantização Pós-Treinamento (Post-Training Quantization)**. O modelo Keras (`.h5`), que utiliza pesos em ponto flutuante de 32 bits (`float32`), foi convertido pela plataforma Edge Impulse para o formato TensorFlow Lite (TFLite).

Durante este processo, foi aplicada a quantização `INT8`, que converte os pesos e ativações do modelo para inteiros de 8 bits. Este passo foi essencial por duas razões:

1. **Redução de Tamanho (Otimização de ROM):** Embora o modelo `float32` já fosse extremamente pequeno (90 KB no formato Keras), a quantização o reduziu ainda mais para **55 KB** (`ei-contador_moedas_final-lite-int.lite`). Isso otimiza o armazenamento e o tempo de carregamento na RAM limitada do dispositivo.
2. **Aceleração da Inferência (Otimização de CPU):** Operações com inteiros (`int8`) são computacionalmente mais eficientes em CPUs de baixo consumo do que operações com ponto flutuante. Esta aceleração foi um fator direto para **superar o requisito de performance de ≥0.5 FPS** no dispositivo-alvo, conforme detalhado na Seção 6.

8.2. Otimização de Software (TFLite Runtime)

Complementando a otimização do modelo, o ambiente de software no Raspberry Pi foi minimizado. Em vez de instalar o pacote completo do TensorFlow, foi utilizado apenas o interpretador **`tflite_runtime`**.

Conforme o *script* de implantação (`interpreter = Interpreter(model_path=model_path)`), esta biblioteca leve gerencia a alocação de tensores e a execução da inferência. Ao reduzir drasticamente as dependências, esta abordagem **evitou potenciais gargalos de consumo de RAM e CPU no nível do sistema operacional**. Como resultado (e discutido na Seção 7), a performance do hardware nunca foi um impedimento, permitindo que a equipe se concentrasse exclusivamente nos desafios de dados e pré-processamento.

- Performance evaluation and analysis - Rodrigo

9. Avaliação de Performance e Análise

Esta seção apresenta os resultados quantitativos e qualitativos do projeto, servindo como a validação final das escolhas de arquitetura (Seção 4) e das técnicas de otimização (Seção 5). A performance foi medida tanto na plataforma de validação (Edge Impulse) quanto no hardware-alvo (Raspberry Pi Zero 2W).

9.1. Performance de Validação (Edge Impulse)

A performance do modelo foi primeiramente validada contra o conjunto de testes (20% do dataset). Este resultado reflete o sucesso direto da estratégia de curadoria de dados discutida na Seção 7.1 (coleta controlada e *augmentation*).

- **Acurácia (F1-Score):** O modelo final alcançou um **F1-Score de [INSERIR VALOR DO F1-SCORE AQUI, ex: 99.0%]**. Este valor excepcionalmente alto confirma que o modelo aprendeu a distinguir com precisão as três classes de moedas sob as condições de iluminação e câmera controladas.

`[INSERIR IMAGEM DA MATRIZ DE CONFUSÃO GERADA PELO EDGE IMPULSE AQUI]`
*(**Figura X**: Matriz de confusão no conjunto de validação, demonstrando mínima confusão entre classes.)*

- **Estimativas de Performance (Pré-implantação):** As estimativas fornecidas pelo Edge Impulse, baseadas no modelo já quantizado para INT8, foram cruciais para validar a viabilidade do hardware *antes* da implantação.
    - **Tempo de Inferência (Estimado):**
    - **Uso de RAM (Estimado):** `[INSERIR USO DE RAM EM KB, ex: 120 KB]`

Estes números confirmaram que a arquitetura leve (MobileNetV2 0.35 + FOMO) era, em teoria, perfeitamente adequada para o Pi Zero 2W.

9.2. Performance de Implantação (Raspberry Pi Zero 2W)

O teste definitivo foi a execução do modelo `.tflite` (INT8) no dispositivo-alvo, utilizando o `tflite_runtime` para processar um *stream* de câmera ao vivo.

Velocidade de Inferência (Quantitativa):

Graças à combinação da arquitetura leve (Seção 4) e da quantização INT8 (Seção 5), a performance no mundo real atendeu e superou as expectativas.

- **Média de FPS (Frames Per Second):** `[INSERIR VALOR DE FPS ALCANÇADO AQUI, ex: 2.5 FPS]`
- **Tempo por Inferência:** `[INSERIR TEMPO EM ms AQUI, ex: 400 ms]`

Este resultado valida que a escolha do modelo e das otimizações foi correta, **superando o requisito mínimo do projeto de ≥0.5 FPS**.

Acurácia Funcional (Qualitativa):

A acurácia no mundo real está diretamente ligada à solução do desafio de pré-processamento (discutido na Seção 7.2).

- **Resultado:** Somente **após** a correção do *script* de inferência para redimensionar a entrada da câmera para **240×240 pixels** (a mesma resolução do treinamento), o modelo passou a funcionar como esperado.
- **Observação:** Com o pré-processamento correto, o sistema se mostrou robusto e preciso, realizando a contagem das moedas em tempo real com uma acurácia funcional equivalente àquela vista no conjunto de validação.

`[INSERIR PRINT/IMAGEM DO SISTEMA EM FUNCIONAMENTO AQUI]` *(**Figura Y**: Demonstração do sistema em tempo real no Pi Zero 2W, identificando e contando moedas corretamente.)*

Conclusão da Análise: A avaliação de performance valida a abordagem completa do projeto. As escolhas de arquitetura (S4) e otimização (S5) entregaram a performance de velocidade de `[INSERIR FPS]`. Mais importante, a performance de acurácia de `[INSERIR F1-SCORE]` demonstra que o sucesso do projeto dependeu diretamente da solução dos desafios de *dados* (S7), especificamente a criação de um dataset controlado e a garantia de consistência no pré-processamento.

- Challenges faced and solutions implemented - Rodrigo

10. Desafios Enfrentados e Soluções Implementadas:

Durante o desenvolvimento do projeto, a equipe encontrou diversos desafios significativos, sendo a maioria relacionada à curadoria do dataset e à transição do modelo treinado para a aplicação no mundo real.

10.1. Desafio: Curadoria do Dataset e "Domain Gap"

O desafio mais crítico do projeto foi criar um dataset que permitisse ao modelo generalizar para as condições reais de operação (câmera do Pi, iluminação do ambiente).

1. **Tentativas Iniciais (Datasets Públicos):**

    - Inicialmente, foi utilizado um dataset público que continha apenas uma moeda por imagem. Este dataset se provou inadequado, pois não representava a complexidade da tarefa (múltiplas moedas sobrepostas).
    - Uma segunda tentativa com outro dataset público (contendo todas as moedas brasileiras) também falhou. Embora o treinamento tenha atingido uma acurácia razoável (≈85%), o modelo não funcionou na prática. A falha foi atribuída ao **"Domain Gap"**: o dataset foi capturado com câmeras e condições de iluminação completamente diferentes das utilizadas no projeto.
2. **Solução Iterativa (Dataset Customizado):**
    - **Primeira Iteração (Coleta em Grupo):** A equipe decidiu criar seu próprio dataset focando em apenas três classes de moedas (1 Real, 50 centavos, 10 centavos) para a Prova de Conceito (POC). Embora as 300 fotos tenham sido tiradas com a câmera correta (Pi Camera), elas foram capturadas "de qualquer jeito", com iluminações variadas e sem um suporte fixo. Isso resultou em um modelo com ≈86% de acurácia, que ainda falhava nos testes ao vivo.
    - **Solução Final (Coleta Controlada + Augmentation):** A solução definitiva foi aplicar um processo de coleta rigoroso. Um único membro da equipe capturou 300 novas imagens usando um **suporte para a câmera** (garantindo ângulos consistentes) e **iluminação constante**. Para enriquecer e robustecer este dataset controlado, foi utilizada a plataforma **Roboflow** para aplicar técnicas de *data augmentation* (rotação, zoom, ruído, etc.), expandindo o dataset para **1.000 imagens**.

Este dataset final (controlado e aumentado) resultou em um modelo com **99% de acurácia** na validação, resolvendo o principal desafio de generalização.

10.2. Desafio: Discrepância de Resolução na Inferência

Mesmo após atingir 99% de acurácia, o modelo falhou nos testes ao vivo. Após investigação, descobriu-se um erro crítico de pré-processamento.

- **Problema:** O modelo foi treinado com imagens de entrada de **240×240 pixels** de imagens capturadas a 320x240. No entanto, o script de inferência ao vivo estava capturando o *frame* da câmera e enviando-o ao modelo em sua resolução nativa 640×480, sem redimensioná-lo.
- **Solução:** O *script* de inferência foi corrigido na etapa de pré-processamento, onde cada *frame* capturado da câmera foi de 320x240 e redimensionado para 240×240 pixels e enviado ao interpretador TFLite. Após essa correção, o modelo passou a funcionar perfeitamente, detectando e contando as moedas com alta precisão.

10.3. Desafio: Ambiente de Hardware (Raspberry Pi)

Devido à experiência prévia adquirida na disciplina, a equipe não enfrentou desafios significativos relacionados à configuração, instalação de dependências ou execução de *scripts* no Raspberry Pi Zero 2W. O foco pôde ser mantido inteiramente nos desafios de Machine Learning.

11. Melhorias Futuras e Extensões:

Embora o projeto tenha alcançado seu objetivo principal como Prova de Conceito (POC), validando a arquitetura FOMO + MobileNetV2 0.35 no Raspberry Pi Zero 2W, diversas melhorias e extensões podem ser exploradas:

1. **Expansão do Número de Classes:**
    - **Inclusão de Todas as Moedas:** O primeiro passo lógico seria expandir o dataset para incluir todas as moedas do Real (R$ 0,01, R$ 0,05, R$ 0,25), aumentando a complexidade da classificação.
    - **Detecção de Notas (Cédulas):** Um desafio maior seria adaptar o sistema para reconhecer também cédulas de dinheiro, o que exigiria uma abordagem diferente (possivelmente segmentação ou classificação de imagem inteira) e um campo de visão maior da câmera.
2. **Melhoria da Robustez do Dataset:**
    - **Variação de Iluminação:** Coletar novos dados sob condições de iluminação adversas (muita luz, pouca luz, sombras) para tornar o modelo menos dependente de um ambiente controlado.
    - **Múltiplos Fundos (Backgrounds):** Treinar o modelo com imagens das moedas sobre diferentes superfícies (madeira, tecido, metal, etc.) para melhorar a generalização.
3. **Otimização de Pós-processamento:**
    - **Rastreamento de Objetos (Tracking):** Em vez de recontar todas as moedas em cada frame, implementar um algoritmo simples de rastreamento (como rastreamento por centroide) para associar detecções entre frames. Isso estabilizaria a contagem total e permitiria ao sistema "lembrar" de moedas mesmo que sejam brevemente ocluídas.
4. **Melhorias na Interface do Usuário (UI/UX):**
    - **Feedback Interativo:** Adicionar recursos à interface web, como um botão para "zerar" a contagem ou salvar o total atual.
    - Deixar mais bonita a interface gráfica.