FACULDADE UNIGUAÇU

IGUAZU SCIENCE

EDITORA UNIVERSITÁRIA UNIGUAÇU

v. 3, n. 7, jun. 2025
ISBN 978-65-83057-12-9

ARTIGO CIENTÍFICO ACESSO LIVRE

APLICAÇÃO DA INTELIGÊNCIA ARTIFICIAL PARA TRADUÇÃO DE LIBRAS

Wagner Dias Menegasso*; Bruno Luiz Schuster Rech**

*Acadêmica de Engenharia de Software - Uniguaçu, wdiasm14@gmail.com.
**Mestre Ciência da Computação – UNIOESTE, *brunolsrech@gmail.com*.

INFORMAÇÕES

Histórico de submissão:

Recebido em: 25 maio. 2025
Aceite: 1 jun. 2025
Publicação *online*: jun. 2025

RESUMO

Este trabalho apresenta o desenvolvimento de uma aplicação baseada em inteligência artificial para a tradução de sinais da Língua Brasileira de Sinais (Libras) para a língua portuguesa. A proposta utiliza algoritmos de visão computacional, com foco no modelo YOLO (You Only Look Once), para detectar e interpretar gestos manuais extraídos de imagens capturadas em tempo real. A aplicação é composta por um sistema embarcado com interface desenvolvida em *Flutter*, uma API de *backend* em FastAPI e um algoritmo pra a criação de um dataset personalizado. O modelo foi treinado com dados personalizados, baseando-se nas posições das mãos por meio de coordenadas de 21 pontos.
Palavras-chave: Libras; Tradução; Dataset; YOLO; Inteligência artificial.

ABSTRACT

This work presents the development of an application based on artificial intelligence for translating Brazilian Sign Language (Libras) into Portuguese. The proposal uses computer vision algorithms, focusing on the YOLO (You Only Look Once) model, to detect and interpret hand gestures extracted from images captured in real time. The application consists of an embedded system with an interface developed in *Flutter*, a *backend* API in FastAPI, and an algorithm for creating a custom dataset. The model was trained with personalized data, based on hand positions through 21-point coordinates.
Keywords: Libras; Translation; Dataset; YOLO; Artificial Intelligence.

Citação: MENEGASSO, Wagner Dias; RECH, Bruno Luiz Schuster. Aplicação da inteligência artificial para tradução de libras. **Iguazu Science**, São Miguel do Iguaçu, v. 3, n. 7, p. 28-36, jun. 2025.

INTRODUÇÃO

A Língua Brasileira de Sinais (Libras) é a principal forma de comunicação utilizada pela comunidade surda no Brasil. Contudo, a barreira de comunicação entre pessoas ouvintes e não ouvintes ainda persiste em diversos contextos sociais. Neste cenário, soluções baseadas em inteligência artificial (IA) e visão computacional têm ganhado destaque como ferramentas promissoras para promover a inclusão.

Este artigo apresenta o desenvolvimento de uma aplicação que visa traduzir sinais de Libras para o português utilizando IA. A proposta se destaca por empregar o modelo You Only Look Once (YOLO), conhecido por sua eficiência na detecção de objetos em tempo real, aliado a um sistema de captura via câmera integrado a uma interface desenvolvida em *Flutter*. O *backend* da aplicação é responsável por processar as imagens e retornar a tradução correspondente, utilizando uma API desenvolvida com FastAPI. Durante o processo de desenvolvimento, identificou-se a necessidade de criar um dataset personalizado, o que motivou a criação de uma aplicação complementar dedicada à coleta, rotulagem e organização dos dados necessários para o treinamento do modelo de reconhecimento de gestos.

O objetivo deste trabalho é demonstrar a viabilidade técnica e prática da utilização de IA, contribuindo com soluções acessíveis e eficientes para a comunicação inclusiva.

METODOLOGIA

O tema demonstra-se de suma importância tendo em vista o advento das tecnologias no âmbito social, e, nesse caso, especificamente na área da engenharia civil. Dessa forma, será realizada uma pesquisa bibliográfica referente ao uso e possíveis usos de tal tecnologia nas avaliações e perícias de imóveis.

Durante a concepção do projeto, o modelo YOLO foi apresentado como sugestão inicial por sua reconhecida eficiência em tarefas de detecção em tempo real. Após uma análise mais aprofundada das variantes disponíveis, o modelo YOLOv11n foi selecionado por oferecer uma boa relação entre desempenho e leveza, características ideais para uma aplicação em dispositivos com recursos limitados. Essa escolha permitiu otimizar o tempo de resposta da aplicação sem comprometer significativamente a acurácia. O modelo foi implementado utilizando a linguagem Python, aproveitando a ampla gama de bibliotecas e suporte da comunidade para tarefas de visão computacional e inteligência artificial.

Figura 1. Métricas de perfomance

Modelo	tamanho (pixéis)	mAPval 50-95	Velocidade CPU ONNX (ms)	Velocidade T4 TensorRT10 (ms)	params (M)	FLOPs (B)
YOLO11n	640	39.5	56.1 ± 0.8	1.5 ± 0.0	2.6	6.5
YOLO11s	640	47.0	90.0 ± 1.2	2.5 ± 0.0	9.4	21.5
YOLO11m	640	51.5	183.2 ± 2.0	4.7 ± 0.1	20.1	68.0
YOLO11l	640	53.4	238.6 ± 1.4	6.2 ± 0.1	25.3	86.9
YOLO11x	640	54.7	462.8 ± 6.7	11.3 ± 0.2	56.9	194.9

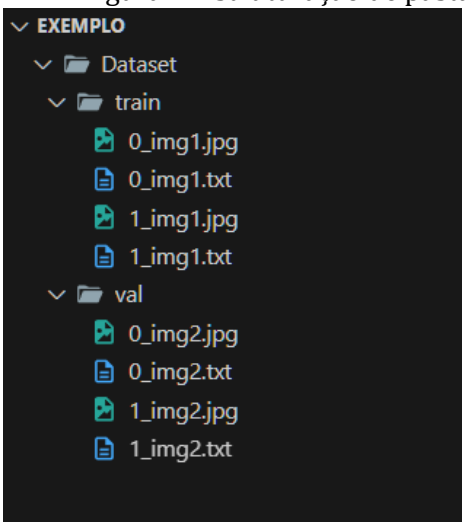
Fonte: Ultralytics

Com base na tabela apresentada na Figura 1, a escolha do modelo YOLOv11n se justifica por seu excelente equilíbrio entre desempenho e eficiência computacional. O modelo se destaca por sua alta velocidade de inferência, especialmente em dispositivos com recursos limitados. Ele é o modelo mais rápido, tanto quando usamos o formato ONNX (Open Neural Network Exchange) levando apenas 56,1 milissegundos para responder, quanto quando é executado em uma placa de vídeo (GPU T4 TensorRT10) leva apenas 1,5 milissegundo. Além disso, é o mais leve entre todos os modelos testados por tem menos "peças internas" (2,6 milhões de parâmetros) e exige menos esforço do computador para funcionar (6,5 bilhões de operações). Essas características o tornam ideal para a aplicações que exigem baixo tempo de resposta, como no caso da proposta desenvolvida, onde a baixa latência e o uso eficiente de recursos são cruciais.

Após a escolha do modelo, foi preciso definir um conjunto de dados adequado para treinar a inteligência artificial. Durante a pesquisa por datasets existentes, percebeu-se que nenhum deles atendia completamente às necessidades do projeto, principalmente na representação dos gestos específicos Libras. Diante disso, optou-se por criar um dataset próprio, o que exigiu o desenvolvimento de uma aplicação complementar voltada para a coleta, rotulagem e organização desses dados. Isso possibilitou um maior controle sobre a qualidade e diversidade dos gestos usados no treinamento, garantindo que o modelo aprendesse a partir de exemplos realmente alinhados com os objetivos da aplicação.

Para a criação da aplicação complementar, foi necessário dividi-la em duas partes principais. A primeira parte ficou responsável pela captura das imagens associadas às classes de sinais, sendo coletadas 100 imagens por classe. Para garantir maior variedade e robustez ao conjunto de dados, essas imagens passam por leves alterações de brilho, contraste e cor. Já a segunda parte cuidou da obtenção da posição, altura e largura da mão dentro do frame da imagem. Para isso, utilizou-se a biblioteca MediaPipe, que detecta 21 pontos na mão. A partir desses pontos, calculou-se a altura sendo as posições dos dois pontos mais distantes no eixo x, e no eixo y a largura. Em seguida, as coordenadas foram normalizadas para ficarem entre 0 e 1, e os dados foram organizados conforme a estrutura de pastas esperada pelo modelo YOLO.

Figura 2. Estruturação de pastas

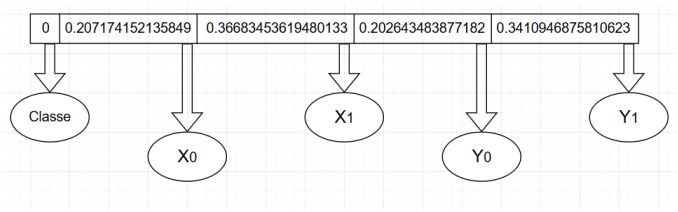


Fonte: Autoria própria

O modelo YOLO exige uma estrutura organizada de pastas e arquivos assim como apresentado na Figura 2, para garantir que o processo de treinamento e validação seja realizado de forma eficiente e padronizada. A separação entre os diretórios train e val permite que o modelo aprenda com um conjunto de dados (treinamento) e seja avaliado com outro (validação), evitando sobreajuste e garantindo uma avaliação realista de desempenho. Cada imagem em formato JPG está acompanhada de um texto em formato TXT com o mesmo nome, que contém as anotações dos objetos (como classe e coordenadas), seguindo o formato esperado pelo YOLO. Essa estrutura facilita a leitura automática dos dados durante o treinamento e é essencial para garantir a

correta associação entre imagens e suas respectivas anotações.

Figura 3. Exemplo de indicação de classes no arquivo TXT



Fonte: Autoria própria

O conteúdo mostrado na Figura 3, representa as anotações no formato YOLO para os objetos detectados nas imagems. Cada linha descreve um objeto identificado na imagem e é composta por cinco valores principais sendo eles respectivamente classe do objeto, duas coordenadas delimitadoras no eixo x e duas no eixo y.

Com o dataset já criado, o próximo passo foi iniciar o treinamento da inteligência artificial. Esse processo, no entanto, tende a ser trabalhoso, pois envolve a definição e o ajuste de diversos hiperparâmetros que afetam diretamente o comportamento do modelo durante o aprendizado. Parâmetros como taxa de aprendizado, número de épocas, tamanho do batch e funções de perda precisam ser ajustados com atenção para equilibrar desempenho e tempo de treinamento. Cada alteração pode gerar resultados diferentes, exigindo testes e comparações constantes até encontrar a configuração mais adequada para atingir bons resultados com os dados disponíveis.

Embora a biblioteca Ultralytics ofereça um recurso que ajusta automaticamente os hiperparâmetros durante o treinamento, optou-se por não utilizá-lo neste projeto. Essa decisão foi motivada por critérios acadêmicos, visando proporcionar um maior controle sobre o processo de aprendizado do modelo e permitir uma compreensão mais profunda do impacto de cada parâmetro nos resultados obtidos.

Com isto, concluiu-se a etapa de ajuste de hiperparametros do modelo de detecção, culminando na configuração exibida na Tabela 1. Nesta etapa, cada parâmetro foi ajustado para otimizar o desempenho do modelo.

Com o modelo treinado, iniciou-se o desenvolvimento da aplicação embarcada utilizando o framework *Flutter*. A escolha por esse framework se deu tanto pela familiaridade prévia com a ferramenta quanto pela sua capacidade de criar aplicações móveis multiplataforma de maneira eficiente. O *Flutter*, que utiliza a linguagem Dart, proporciona uma estrutura baseada em widgets reutilizáveis e um gerenciamento de estado claro, o que facilita a construção de interfaces dinâmicas e responsivas. Esses fatores foram decisivos para a criação de um aplicativo funcional, leve e com boa experiência de uso, capaz de capturar imagens da câmera do dispositivo e interagir com o *backend* para a tradução em tempo real dos sinais capturados.

Tabela 1. Hiperparametros do modelo de classificação

Parâmetros	Valores	Função
Epochs	65	Número total de épocas
Batch	16	Tamanho do *batch* para cada iteração
Imgsz	650	Tamanho da imagem usada no treinamento
Device	Cuda	Dá preferência ao uso da GPU (se houver)
lr0	0,006	Taxa de aprendizado inicial
Lrf	0,06	Fator final da taxa de aprendizado
Momentum	0,937	Fator de momentum do otimizador
weight_decay	0,0006	Penalidade de regularização para evitar overfitting
Optimizer	SGD	Algoritmo de otimização utilizado
Workers	8	Número de subprocessos para carregamento de dados
Patience	15	Número de épocas sem melhoria antes de parar o treinamento
hsv_h	0,015	Ajuste de matiz para aumento de dados (hue)
hsv_s	0,7	Ajuste de saturação para aumento de dados
hsv_v	0,4	Ajuste de valor (brilho) para aumento de dados
Mosaic	1	Probabilidade de aplicar a técnica de mosaic para aumento de dados

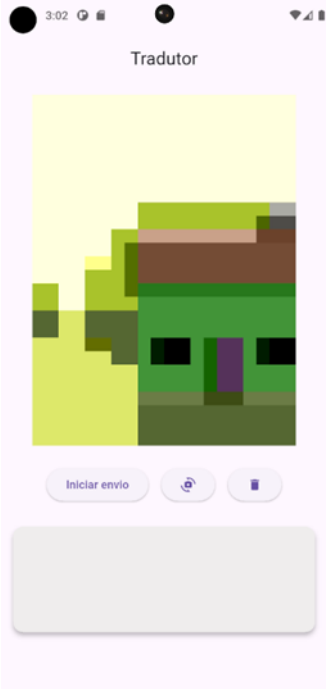
Fonte: Autoria própria

Para atender às necessidades específicas do projeto, foi idealizada uma aplicação que prezasse pela simplicidade e facilidade de uso. A proposta consiste em um sistema que captura imagens diretamente da câmera do dispositivo móvel, envia essas imagens por meio de uma API para um servidor contendo o modelo de IA treinado, e então recebe como resposta o sinal correspondente traduzido. Esse processo torna a aplicação mais leve e eficiente, já que o processamento pesado é feito no servidor, permitindo uma resposta rápida e precisa na tradução dos sinais. A integração entre o aplicativo e o *backend* foi fundamental para garantir a fluidez da experiência do usuário.

Na Figura 4 está a representação da aplicação mobile que conta com uma página inicial intitulada

"Tradutor", onde é exibida a visualização da câmera do dispositivo e três botões com funções distintas. O primeiro botão, denominado "iniciar envio", tem como objetivo iniciar uma sequência de requisições à API para tradução dos sinais, ao ser pressionado novamente, essa sequência é interrompida. O segundo botão, representado por um ícone de câmera, permite alternar entre a câmera traseira e frontal do dispositivo. Por fim, o botão com ícone de lixeira apaga o texto exibido na tela, permitindo ao usuário reiniciar o processo. Abaixo desses botões, há um campo dedicado à exibição dos resultados retornados pela API, apresentando os sinais interpretados a partir das imagens capturadas pela câmera.

Figura 4. Tela da aplicação mobile



Fonte: Autoria própria

Passando então para a criação de uma API utilizando o FastAPI. O FastAPI é um framework moderno e de alto desempenho para a construção de APIs com Python, conhecido por sua simplicidade, rapidez e por utilizar tipos de dados para validação automática. A escolha dessa ferramenta se deu pela facilidade de integração com o modelo treinado, bem como pela boa performance e escalabilidade oferecidas.

A estrutura da API é dividida em duas partes, a primeira foi implementada em Python, utilizando o FastAPI, e disponibiliza um servidor na seguinte URL: "http://<ip da máquina>:8000/predict". Esse endpoint recebe uma imagem enviada via método POST e retorna uma predição em formato JSON, com uma chave chamada "resultado" contendo o valor identificado pelo modelo treinado. A segunda etapa acontece no aplicativo mobile desenvolvido com *Flutter*, que realiza uma requisição HTTP para essa URL, enviando a imagem capturada pela câmera do dispositivo. Ao receber a resposta da API, o aplicativo extrai o valor correspondente do JSON e exibe o sinal interpretado na interface do usuário.

Na Figura 5, está representada a lógica utilizada na API desenvolvida para processar as imagens capturadas pela aplicação móvel e retornar o resultado gerado pelo modelo treinado.

Figura 5. Lógica do servidor

```python
app = FastAPI()
model = YOLO("runs/detect/train5/weights/best.pt")

sinal_sequencial = {}

@app.post("/predict")
async def predict(file: UploadFile = File(...)):
    contents = await file.read()
    npimg = np.frombuffer(contents, np.uint8)
    frame = cv2.imdecode(npimg, cv2.IMREAD_COLOR)

    results = model(frame)

    for r in results:
        class_indices = r.boxes.cls.int().tolist()
        class_names = [model.names[idx] for idx in class_indices]

        for s in class_names:
            return JSONResponse(content={"resultado": s})

    return JSONResponse(content={"resultado": 'null'})
```

Fonte: Autoria própria

A Figura 6 ilustra o fluxo completo de envio e recebimento de imagens via API. Esse processo começa na aplicação desenvolvida em *Flutter*, onde o usuário aciona o botão para iniciar o envio de imagens capturadas pela câmera do dispositivo. Cada imagem é convertida e enviada para a API por meio de uma requisição HTTP do tipo POST. No *backend*, a API construída com FastAPI recebe a imagem, decodifica-a utilizando OpenCV e a processa com o modelo YOLO previamente treinado.

Por fim, durante a criação do projeto, foi observado que alguns sinais da Língua Brasileira de Sinais (Libras) exigem movimento contínuo da mão para serem corretamente compreendidos, como é o caso da letra "J". Como a base do projeto utiliza o modelo YOLO, que é voltado para a identificação de objetos em imagens estáticas, há uma limitação em reconhecer sinais que dependem de movimento. Apesar do modelo possuir funcionalidades de rastreamento, ele não associa por si só a movimentação ao significado de uma classe.

Tendo o problema da letra "J" em mente, foi desenvolvida uma lógica alternativa para possibilitar o reconhecimento de sinais que envolvem movimento contínuo. Essa abordagem consiste em dividir o gesto

dinâmico em múltiplas etapas estáticas, representadas por diferentes classes. No caso da letra "J", o movimento foi segmentado em quatro fases distintas, nomeadas como "j1*", "j2*", "j3*" e "j4*", correspondendo respectivamente ao início, meio e fim da trajetória feita pela mão. O caractere especial "*" foi adicionado ao final das classes que representam sinais com movimento, permitindo que sejam identificadas e tratadas de forma diferenciada durante o processamento. Essa estratégia possibilita que, mesmo utilizando um modelo voltado para imagens estáticas, como o YOLO, seja viável interpretar movimentos complexos por meio da detecção sequencial dessas etapas, aproximando-se do comportamento esperado na linguagem de sinais.

Figura 6. Lógica do cliente

```dart
import 'package:http/http.dart' as http;
Future<void> _sendFrame() async {
 if (!_controller!.value.isInitialized) return;
 try {
  await _controller!.takePicture().then((file) async {
   final imageFile = File(file.path);
   var request = http.MultipartRequest(
    'POST',
    Uri.parse('http://192.168.0.194:8000/predict'),
   );
   request.files.add(await http.MultipartFile.fromPath('file', imageFile.path));
   var response = await request.send();
```

Fonte: Autoria própria

Na Figura 7 está a lógica implementada para solucionar o problema das letras caracterizadas por movimento. Essa lógica analisa uma sequência de quatro requisições consecutivas ao modelo, verificando se pelo menos três dessas requisições possuem classes que iniciam com a mesma letra base, desconsiderando o caractere especial "*". Caso essa condição seja atendida, considera-se que houve detecção consistente do movimento correspondente àquela letra, e o resultado final é validado como sendo a própria letra em questão.

Essa abordagem aumenta a confiabilidade do reconhecimento, filtrando variações pontuais e garantindo que apenas movimentos coerentes e contínuos sejam interpretados como sinais válidos.

Figura 7. Lógica de movimento

```dart
if (result != 'null' && result!.isNotEmpty && result!.contains('*')){
      setState(() {
        _sequencialResult.add(result!);
        var counts = <String, int>{};
        for (var s in _sequencialResult) {
          counts[s[0]] = (counts[s[0]] ?? 0) + 1;
        }
        String? repeated;
        counts.forEach((key, value) {
          if (value >= 3) {
          repeated = key;
          }
        });
        if (repeated != null) {
          accumulatedResult += repeated!.substring(0, repeated!.length - 2);
          _sequencialResult.clear();
          _success = true;
        }
      });
    }
```

Fonte: Autoria própria

RESULTADOS E DISCUSSÃO

Nesta seção, serão discutidos os resultados obtidos com o uso desse modelo, destacando seu desempenho, bem como os principais desafios enfrentados durante o desenvolvimento do projeto, como falhas na diferenciação de sinais semelhantes e dificuldade na detecção em ângulos não padronizados.

Como já mencionado anteriormente, o projeto utiliza um modelo voltado para a identificação de objetos em imagens estáticas, mais especificamente um modelo da família YOLO. Embora seja eficiente para detectar e localizar objetos com rapidez e precisão, esse tipo de modelo não foi originalmente desenvolvido para interpretar sinais gestuais complexos, como é o caso da linguagem Libras. Essa limitação se torna evidente ao lidar com gestos que exigem movimentação contínua ou variações sutis de posição, como ocorre entre letras como "f" e "t" que são diferenciadas apenas pela posição do polegar. Por estas razões, foi optado por fazer a detecção somente de letras do alfabeto onde podemos ter um controle melhor tanto do treinamento quanto dos resultados esperados, assim podendo usar este projeto como método de estudo e um ponto de partida para um projeto futuro.

Voltando a um problema já mencionado anteriormente, relacionado às letras que exigem movimento para serem compreendidas corretamente, foi possível implementar uma solução parcial. A estratégia adotada, que consiste em dividir a movimentação em uma sequência de quatro classes temporais, como no caso da letra "j" representada por "j1*", "j2*", "j3*" e "j4*", mostrou-se eficaz para esse tipo de sinal. No entanto, ao aplicar essa mesma lógica à letra "z", surgiram novas dificuldades. O gesto correspondente à letra "z" em Libras não apresenta

mudanças significativas de angulação ou variações perceptíveis entre as etapas da movimentação. Como consequência, torna-se extremamente difícil distinguir as fases "z1*", "z2*", "z3*" e "z4*" de forma confiável, mesmo utilizando imagens em sequência. Diante dessa limitação e da ausência de uma solução prática e funcional durante o desenvolvimento do projeto, optou-se por não incluir a letra "z" entre os sinais reconhecidos pelo modelo, uma vez que não foi possível contornar essa dificuldade de maneira satisfatória.

Quanto à questão do dataset, alguns desafios foram enfrentados durante sua construção, principalmente relacionados à detecção dos 21 pontos da mão feita pela biblioteca externa utilizada, o MediaPipe. Em certas situações, essa detecção falhava, especialmente quando havia movimentos bruscos, iluminação inadequada ou quando a mão se posicionava em ângulos que dificultavam a identificação dos pontos de referência.

Como a biblioteca em si não oferecia mecanismos para correção direta dessas falhas, a solução encontrada foi adotar maior cautela durante o processo de captura, realizando os gestos de forma mais controlada e estável, o que ajudou a minimizar os erros e aumentar a taxa de detecção bem-sucedida. Apesar desse cuidado, ainda ocorriam falhas, e nestes casos foi necessário recorrer à marcação manual das classes, o que tornou o processo mais demorado e sujeito a erros humanos.

Essa etapa evidenciou a importância de um ambiente controlado para a coleta de dados e apontou a necessidade de melhorias futuras no sistema de captura, como a implementação de mecanismos de verificação da integridade dos pontos ou a substituição da ferramenta por alternativas mais robustas. Mesmo com essas limitações, foi possível compor um dataset funcional, que serviu como base para o treinamento e testes do modelo.

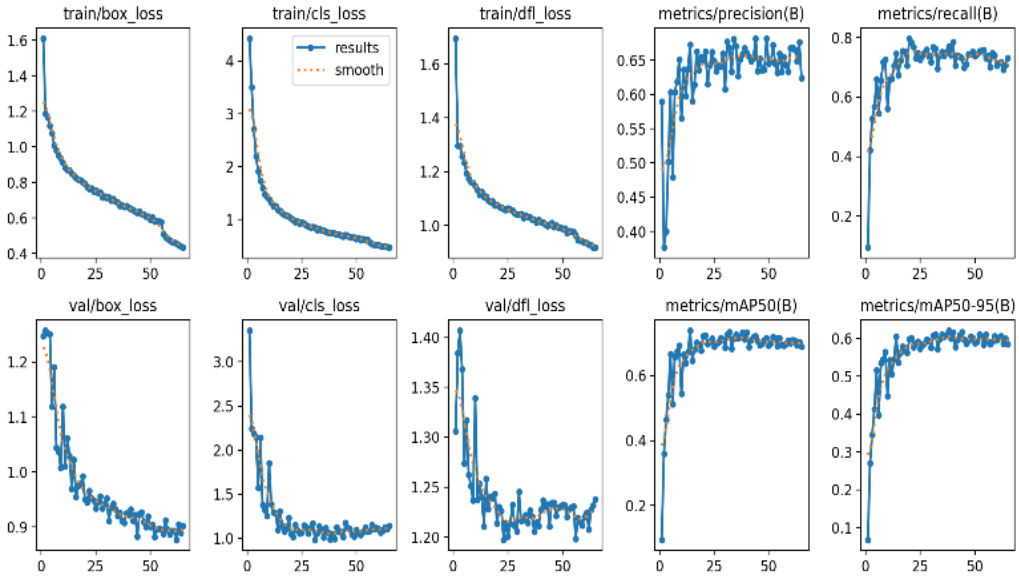
Em relação à etapa de treinamento, mesmo com o uso de uma GPU dedicada para o processamento dos dados, o que proporcionou uma redução significativa no tempo de treinamento em comparação ao uso de CPU, o processo de refinamento dos hiperparâmetros ainda foi bastante demorado. Isso ocorreu principalmente devido ao tamanho do dataset utilizado.

Apesar do foco estar apenas no alfabeto, foram incluídas 100 imagens para cada uma das 24 letras detectadas, com exceção da letra "z", além de um conjunto especial de 400 imagens destinado à letra "j", que foi dividida em quatro classes temporais ("j1*", "j2*", "j3*", "j4*"). Com isso, o total de imagens utilizadas chegou a 2.800, tornando o processo de ajuste fino dos hiperparâmetros mais trabalhoso e exigente. A complexidade aumentou ainda mais com a necessidade de garantir a generalização do modelo e evitar overfitting, o que exigiu múltiplos ciclos de teste e validação ao longo do desenvolvimento.

Apesar dos contratempos enfrentados durante o desenvolvimento, como falhas na detecção de pontos, limitações na identificação de sinais com movimento e o tempo elevado necessário para o refinamento dos hiperparâmetros, foi possível alcançar um resultado satisfatório dentro dos objetivos propostos.

Como mostrado na Figura 8, durante o treinamento do modelo YOLO, foi possível observar uma evolução consistente em diversas métricas, tanto no conjunto de treinamento quanto no de validação. As perdas associadas à localização dos objetos (box_loss), classificação (cls_loss) e regressão refinada das caixas (dfl_loss) apresentaram quedas contínuas ao longo

Figura 8. Resultado do treinamento



Fonte: Autoria própria

das épocas, evidenciando que o modelo estava aprendendo de forma progressiva e estável. As métricas de precisão e recall também demonstraram bons resultados, atingindo patamares acima de 0.6 e 0.7, respectivamente, o que indica que o modelo foi eficiente em identificar corretamente os sinais e reduzir a quantidade de falsos positivos e negativos. As métricas de média de precisão (mAP50 e mAP50-95) reforçam esse bom desempenho, ultrapassando a marca de 0.6, valor considerado satisfatório para o escopo deste projeto. Apesar de algumas oscilações pontuais, principalmente nas métricas relacionadas à validação, o comportamento geral foi positivo. Esses resultados indicam que, mesmo com as limitações inerentes ao modelo YOLO no reconhecimento de gestos com movimento contínuo, como a letra "z", o sistema mostrou-se eficaz para detecção de sinais estáticos e movimentos simples, validando sua aplicação como ferramenta de estudo.

Em relação à velocidade de transferência de dados entre o modelo treinado e a aplicação, os resultados foram satisfatórios dentro do contexto do projeto, embora ainda que o ideal para aplicações como essa o tempo de resposta tende a ser menor. Mesmo operando por meio de uma API local, cada requisição leva aproximadamente 1 segundo para ser processada, o que pode comprometer a fluidez da interação em cenários que exigem respostas mais imediatas. Inicialmente, toda a lógica de interpretação dos resultados gerados pelo modelo era executada no lado do servidor, o que contribuía para o tempo total de resposta.

Buscando otimizar esse aspecto, foi realizada uma alteração estrutural: a lógica de tratamento dos dados foi transferida para o lado do cliente, ficando o servidor responsável apenas por retornar os dados brutos gerados pelo modelo. No entanto, apesar dessa tentativa de otimização, a redução no tempo de resposta não foi significativa, indicando que o gargalo está possivelmente relacionado ao tempo de inferência do próprio modelo ou ao processo de comunicação entre as partes da aplicação.

Outro fator que pode ter impactado o desempenho é o tamanho do modelo utilizado, uma vez que modelos mais complexos e robustos, como os baseados em YOLO, demandam mais recursos computacionais, mesmo após o treinamento. Além disso, o formato e a quantidade de dados transmitidos a cada requisição também podem influenciar diretamente o tempo de resposta. Ainda que o uso de uma API local elimine a latência associada à internet, a comunicação entre a aplicação cliente e o servidor pode sofrer impactos por limitações de hardware ou pela forma como os dados estão sendo tratados e serializados.

Para a aplicação desenvolvida em *Flutter*, cuja função principal é tratar e exibir os dados retornados da API, optou-se por uma abordagem simples e direta, com o objetivo de manter o foco do projeto voltado à análise e desempenho do modelo de inteligência artificial. Essa simplicidade teve como propósito facilitar o uso da aplicação, reduzir a complexidade geral do sistema e manter o escopo do projeto bem definido, uma vez que o desenvolvimento da interface em si não era o objetivo central.

Durante a criação da aplicação, não foram encontradas dificuldades significativas no desenvolvimento da interface ou na estruturação da lógica interna, com exceção da etapa de integração com a API. Essa parte apresentou alguns desafios técnicos, especialmente por se tratar de uma aplicação mobile, onde há particularidades na comunicação entre o frontend e o *backend*, como o manuseio de requisições HTTP, permissões de rede e o tratamento de CORS (Cross-Origin Resource Sharing), que é um mecanismo de segurança dos navegadores que restringe requisições feitas entre diferentes origens. Ainda assim, com as devidas configurações e ajustes no código, foi possível estabelecer uma comunicação funcional entre a aplicação e o servidor, garantindo a exibição correta dos dados processados.

A imagem da Figura 9 apresenta uma demonstração dos resultados obtidos com a detecção em tempo real dos sinais. Nessa demonstração, o objetivo foi representar todo o alfabeto em ordem sequencial, a fim de verificar a capacidade do modelo em identificar corretamente cada letra em uma situação prática. Embora o modelo não tenha alcançado uma taxa de acerto perfeita, os resultados foram considerados satisfatórios, especialmente quando levamos em conta os diversos contratempos enfrentados ao longo do desenvolvimento, como as limitações do modelo YOLO para capturar movimentos, falhas ocasionais na detecção dos pontos da mão e os desafios enfrentados na construção e refinamento do dataset.

Figura 9. Uso da aplicação

Fonte: Autoria própria

Analisando o resultado do experimento, é possível observar alguns erros de detecção que ocorreram com certa frequência. Entre eles, destaca-se a troca da letra "f" pela letra "t", da letra "s" pela letra "a" e da letra "v" pela letra "w", além da duplicidade da letra "i". Ao observar as posições das mãos associadas a essas letras, nota-se que elas possuem grande semelhança visual, o que pode ter confundido o modelo durante o processo de classificação, levando a erros compreensíveis considerando as limitações da arquitetura utilizada.

No caso específico da duplicidade da letra "i", uma possível explicação é que o movimento inicial da letra "j" representado pela classe "j1*" possui uma posição estática muito semelhante à da letra "i". Dessa forma, durante a execução do gesto para a letra "j", o modelo pode ter identificado a sequência temporal como sendo ("i", "j2*", "j3*", "j4*"). Como a lógica de validação exige que pelo menos três das quatro classes temporais comecem com a mesma letra para confirmar o movimento, o sistema acaba reconhecendo tanto a letra "i" quanto a "j" na mesma sequência, gerando a duplicidade observada.

A Figura 10 apresenta um teste de assertividade feito com o objetivo de avaliar o desempenho do modelo em relação às letras que apresentaram maior índice de erros durante os testes anteriores. Para isso, foram realizadas quatro tentativas para cada uma das letras de difícil distinção, no caso "f", "s", "v". Os resultados mostraram que, embora o modelo tenha sido capaz de reconhecer corretamente algumas dessas letras em determinadas posições ou ângulos, a taxa de acerto foi significativamente inferior se comparada às demais letras do alfabeto.

Figura 10. Teste de assertividade

Fonte: Autoria própria

Isso reforça a hipótese levantada anteriormente de que as similaridades visuais entre certos sinais, associadas às limitações do modelo YOLO na detecção de gestos sutis ou sobrepostos, dificultam a distinção entre sinais com características muito próximas. Dessa forma, o teste evidencia que, apesar de o modelo ter uma capacidade razoável de acerto, ele ainda apresenta dificuldades relevantes em contextos em que a precisão na diferenciação entre gestos similares é essencial.

Com o fortalecimento da hipótese apresentada, confirma-se o que já era esperado em relação ao desempenho do modelo. Por se tratar de um sistema baseado na detecção de objetos a partir de imagens estáticas, o modelo analisa a mão como se fosse apenas mais um objeto a ser classificado, sem considerar sua anatomia, posição relativa dos dedos ou contexto temporal dos movimentos. Isso limita significativamente sua capacidade de distinguir gestos semelhantes ou identificar nuances sutis entre sinais, especialmente em uma linguagem visual como a Libras, onde pequenas variações na forma e na orientação podem alterar completamente o significado do sinal.

Planejamento para Aplicações Futuras

Apesar das complicações enfrentadas durante a confecção do projeto, a experiência adquirida ao longo do desenvolvimento proporcionou um aprendizado significativo que poderá ser aplicado em iniciativas futuras. Compreendeu-se melhor as limitações de utilizar uma arquitetura baseada em detecção de objetos, como o YOLO, para interpretar sinais de Libras. Para um próximo projeto, pretende-se explorar arquiteturas mais adequadas ao contexto, que possibilitam a análise de sequências de imagens ao longo do tempo, captando melhor os aspectos dinâmicos dos sinais. Ainda assim, não se descarta o uso da arquitetura YOLO como parte complementar da solução, dada sua eficiência na identificação de gestos estáticos e sua velocidade de inferência. A combinação entre diferentes abordagens pode levar a um sistema mais robusto e preciso, aproximando ainda mais a tecnologia das necessidades reais de acessibilidade e comunicação da comunidade surda.

CONCLUSÕES

Este projeto teve como principal objetivo estudar e aplicar o processo de treinamento de um modelo de detecção de sinais da Língua Brasileira de Sinais, utilizando como base a arquitetura YOLO. Durante o desenvolvimento, diversos desafios foram enfrentados, principalmente em relação à limitação do modelo em interpretar gestos que envolvem movimento, como as letras "J" e "Z", além das dificuldades na construção de um dataset consistente

devido à sensibilidade da biblioteca de detecção de pontos. Ainda assim, com adaptações criativas, como a divisão de gestos em classes temporais e o uso de caracteres especiais para diferenciá-los, foi possível contornar parcialmente essas limitações e alcançar resultados satisfatórios.

O modelo apresentou desempenho aceitável dentro do escopo proposto, permitindo identificar corretamente a maioria das letras estáticas do alfabeto em tempo real. A aplicação *Flutter*, embora simples, cumpriu bem seu papel ao integrar-se com a API e exibir os resultados de forma funcional. Os testes demonstraram que há margem para melhorias, especialmente na distinção de gestos semelhantes e na precisão temporal, mas também evidenciaram o potencial dessa abordagem como base para estudos futuros.

Portanto, mesmo com suas limitações, o projeto cumpriu seu propósito pedagógico, fornecendo aprendizados valiosos sobre o funcionamento de modelos de visão computacional, tratamento de dados de entrada, e os desafios práticos de trabalhar com linguagem de sinais. Como continuidade, recomenda-se explorar modelos mais adequados para sequências temporais, como redes neurais recorrentes (RNNs) ou Transformers, além da utilização de bibliotecas ou sensores que ofereçam maior robustez na detecção de movimentos e posições da mão.

REFERÊNCIAS

ALMEIDA, João Carlos de. **Visão computacional aplicada ao reconhecimento de gestos**. São Paulo: Novatec, 2020.

BOCHENEK, Michal *et al.* Real-time Hand Gesture Recognition using YOLOv5. In: Proceedings of the 2022 IEEE/CVF **Conference on Computer Vision and Pattern Recognition**. 2022. Disponível em: https://arxiv.org/abs/2205.14459. Acesso em: 20 maio 2025.

FONSECA, Luana R.; MENEZES, Pedro H. Reconhecimento de sinais da Libras com aprendizado profundo. **Revista Brasileira de Computação Aplicada**, São Paulo, v. 13, n. 1, p. 18–31, 2021.

REDMON, Joseph; FARHADI, Ali. **YOLOv3: An Incremental Improvement**. 2018. Disponível em: https://arxiv.org/abs/1804.02767. Acesso em: 20 maio 2025.

SILVA, Marcos A. da. **Processamento de Imagens e Reconhecimento de Padrões**. 2. ed. Rio de Janeiro: LTC, 2019.

ASSALEH, K.; SHANABLEH, T.; HAJJAJ, H. Recognition of handwritten Arabic alphabet via hand motion tracking. **Journal of the Franklin Institute**, v. 346, p. 175–189, 2009. DOI: 10.1016/j.jfranklin.2008.08.005.

JAIN, R.; KARSH, R. K.; BARBHUIYA, A. A. Encoded motion image-based dynamic hand gesture recognition. **The Visual Computer**, v. 38, p. 3305–3324, 2021. DOI: 10.1007/s00371-021-02259-3.

PAHLEVANZADEH, M.; VAFADOOST, M.; SHAHNAZI, M. **Sign language recognition**. In: 2007 3rd International Conference on Intelligent Systems and Applications. IEEE, 2007. p. 1–6. DOI: 10.1109/ISAP.2007.4441592.

ZHU, Y.; YANG, Z.; YUAN, B. Vision based hand gesture recognition. In: 2013 **International Conference on Service Science.** IEEE, 2013. p. 260–265. DOI: 10.1109/ICSS.2013.40.