

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE ENGENHARIA ELÉTRICA
ENGENHARIA ELETRÔNICA E DE TELECOMUNICAÇÕES
CAMPUS PATOS DE MINAS

ARTHUR NAZÁRIO DA COSTA

**UMA ANÁLISE DETALHADA DO
FUNCIONAMENTO DE QR CODES**

Patos de Minas - MG

2025

ARTHUR NAZÁRIO DA COSTA

UMA ANÁLISE DETALHADA DO FUNCIONAMENTO DE QR CODES

Trabalho de conclusão de curso apresentado à Faculdade de Engenharia Elétrica, da Universidade Federal de Uberlândia, Campus Patos de Minas, Minas Gerais, como requisito parcial à obtenção da graduação em Engenharia Eletrônica e de Telecomunicações.
Orientador: Prof.^a Dr.^a Eliana Pantaleão

Patos de Minas - MG

2025

ARTHUR NAZÁRIO DA COSTA

UMA ANÁLISE DETALHADA DO FUNCIONAMENTO DE QR CODES

Trabalho de conclusão de curso apresentado à Faculdade de Engenharia Elétrica, da Universidade Federal de Uberlândia, Campus Patos de Minas, Minas Gerais, como requisito parcial à obtenção da graduação em Engenharia Eletrônica e de Telecomunicações.

Prof.^a Dr.^a Eliana Pantaleão
Orientador

Me. Daniel de Oliveira Ferreira
Coorientador

Prof. Dr. Rafael Augusto da Silva
Examinador

**Prof. Dr. Laurence Rodrigues do
Amaral**
Examinador

Dedico este trabalho a minha família, em especial ao meus pais Hέλvio Nazάrio dos Santos e Lilian Cristina da Costa Santos que me incentivaram durante minha jornada, e aos meus amigos e companheiros.

Agradecimentos

Agradeço primeiramente aos meus pais que me incentivaram aos estudos e a seguir em frente nos momentos mais difíceis da minha vida.

Agradeço também ao corpo docente por me incentivar a trilhar este caminho e enxergar um potencial que eu não enxergava ao ingressar na graduação.

Sou bastante grato também a todos os meus amigos e companheiros por acompanharem meu desenvolvimento pessoal e profissional dentro da universidade.

Por fim, dedico este Projeto Final de Curso a todos os meus colegas de trabalho da Obers Sistemas que me deram a mão para o meu ingresso ao mercado de trabalho.

Resumo

Os *QR codes* (*Quick Response Code*) são amplamente utilizados para armazenar e compartilhar informações em aplicações como pagamentos digitais, rastreamento de produtos e divulgação de conteúdo. Apesar de sua ampla utilização, diversos usuários desconhecem a estrutura interna e os processos envolvidos na codificação e decodificação das informações presentes em tais códigos. O objetivo do trabalho é apresentar uma análise do funcionamento dos *QR codes*, abordando sua estrutura, os mecanismos de codificação de dados e níveis de correção de erros empregados para garantir a integridade das informações. Para apoiar a compreensão desses processos, foi desenvolvido um aplicativo capaz de codificar e decodificar *QR codes* utilizando a câmera de dispositivos móveis. O sistema foi construído com auxílio de bibliotecas de visão computacional e integrado a uma aplicação *mobile* desenvolvida em React Native. Além disso, foram realizados experimentos para avaliar o desempenho do sistema em diferentes condições de leitura, considerando fatores como distância, iluminação e rotação do código. Os resultados obtidos demonstram a capacidade do aplicativo em reconhecer *QR Codes* de diferentes versões e níveis de correção de erro, além de evidenciar limitações do processo de detecção em condições adversas. Dessa forma, o trabalho contribui para a compreensão do funcionamento dos *QR Codes*, além de apresentar uma ferramenta prática para demonstrar os processos de codificação e leitura desses códigos.

Palavras-chave: QR Code; visão computacional; codificação de dados; decodificação; aplicativos móveis.

Abstract

QR codes (Quick Response Codes) are widely used to store and share information in applications such as digital payments, product tracking, and content dissemination. Despite their widespread use, many users are unaware of the internal structure and processes involved in encoding and decoding the information contained in these codes. The aim of this work is to present an analysis of how QR codes work, addressing their structure, data encoding mechanisms, and error correction levels employed to ensure information integrity. To support the understanding of these processes, an application capable of encoding and decoding QR codes using the camera of mobile devices was developed. The system was built with the aid of computer vision libraries and integrated into a mobile application developed in React Native. In addition, experiments were conducted to evaluate the system's performance under different reading conditions, considering factors such as distance, lighting, and code rotation. The results obtained demonstrate the application's ability to recognize QR codes of different versions and error correction levels, as well as highlighting limitations in the detection process under adverse conditions. In this way, the work contributes to the understanding of how QR Codes work, in addition to presenting a practical tool to demonstrate the encoding and reading processes of these codes.

Keywords: QR Code; computer vision; data encoding; decoding; mobile applications.

Lista de ilustrações

Figura 2.1 – Diagrama de blocos do funcionamento do <i>QR code</i>	19
Figura 2.2 – Padrão de detecção.	20
Figura 2.3 – Separador.	21
Figura 2.4 – Padrão de temporização.	21
Figura 2.5 – Padrão de alinhamento.	22
Figura 2.6 – Localização do formato a informação.	23
Figura 2.7 – Informação da versão.	24
Figura 2.8 – Zona silenciosa.	24
Figura 2.9 – Diferença entre as versões do <i>QR code</i>	25
Figura 2.10– <i>QR code</i> nos modelos 1 e 2.	26
Figura 2.11–Micro <i>QR code</i>	26
Figura 2.12–Versão <i>frame</i>	27
Figura 2.13– <i>iQR code</i>	28
Figura 2.14–Posição dos <i>bits</i> de tipo de dado	30
Figura 2.15–Elementos de correção de erro	32
Figura 2.16–Distribuição dos <i>bytes</i>	34
Figura 2.17–Distribuição dos <i>bytes</i> na horizontal	34
Figura 2.18–Caminho dos <i>bits</i>	35
Figura 3.1 – Modelo de camadas SSD	40
Figura 3.2 – Bloco Residual vs Bloco Residual Invertido	49
Figura 3.3 – Evolução dos blocos de convolução separável	50
Figura 4.1 – Diagrama decodificação do sistema	56
Figura 4.2 – Diagrama codificação do sistema	57
Figura 4.3 – Montagem do experimento	61
Figura 5.1 – Resultados obtidos do experimento	64
Figura 5.2 – <i>QR codes</i> da versão 40 reconhecidos pelo aplicativo na altura mínima	65
Figura 5.3 – <i>QR codes</i> das versões 1, 10 e 20 detectados nas condições mais adversas	66
Figura 5.4 – <i>QR codes</i> da versão 1 reconhecidos pelo aplicativo na altura máxima	67
Figura 5.5 – <i>Print</i> da tela do aplicativo com o <i>QR code</i> versão 40 ECC M não reconhecido em um ângulo de 10°	68
Figura 5.6 – <i>QR codes</i> versão 40 reconhecidos no ângulo de 10°	69

Figura 5.7 – <i>Print</i> da tela do aplicativo com o <i>QR code</i> versão 40 ECC H não reco-	
nhecido em um ângulo de 20°	70
Figura 5.8 – Inserção das requeridas informações na seção de textos do aplicativo . .	71
Figura 5.9 – <i>Link</i> codificado pelo aplicativo	72
Figura 5.10–Textos codificados pelo aplicativo	73

Lista de tabelas

Tabela 2.1 – Tabela de coordenadas dos padrões de alinhamento	22
Tabela 2.2 – Tabela comparativa da capacidade de armazenamento entre as versões do Micro <i>QR code</i>	27
Tabela 2.3 – Capacidade de dados do tipo numérico e alfanumérico por nível de correção de erro	29
Tabela 2.4 – Representação binária dos tipos de dados	30
Tabela 2.5 – Codificações alfanuméricas	31
Tabela 2.6 – Níveis de correção de erro	32
Tabela 2.7 – Posições dos <i>bytes</i>	36
Tabela 3.1 – Bloco residual de gargalo (<i>bottleneck</i>) transformando de k para k' ca- nais, com salto (<i>stride</i>) s e fator de expansão t	49
Tabela 3.2 – Descrição da arquitetura da rede <i>MobileNetV2</i>	50
Tabela 4.1 – Dados utilizados nos testes de codificação do aplicativo	59
Tabela 4.2 – Especificações técnicas da câmera do celular Samsung Galaxy S20 . . .	62

Lista de abreviaturas e siglas

AI	Inteligência Artificial (<i>Artificial Intelligence</i>)
ANN	Rede Neural Artificial (<i>Artificial Neural Network</i>)
APP	Aplicativo (<i>application</i>)
BBOX	Caixa Delimitadora (<i>Bounding Box</i>)
CNN	Rede Neural Convolutacional (<i>Convolutional Neural Network</i>)
DSC	Convoluções Separáveis de Profundidade (<i>Depthwise Separable Convolutions</i>)
ECC	Código de Correção de Erro (<i>Error Correction Code</i>)
IDE	Ambiente de Desenvolvimento Integrado (<i>Integrated Development Environment</i>)
IOS	Sistema Operacional do Iphone (<i>Iphone Operating System</i>)
IoU	Intersecção sobre a União (<i>Intersection over Union</i>)
QR code	Código de Resposta Rápida (<i>Quick Response Code</i>)
SQRC	Código de Resposta Rápida Seguro (<i>Secure Quick Response Code</i>)
SSD	Detector MultiBox de Disparo Único (<i>Single Shot MultiBox Detector</i>)
URI	Identificador Uniforme de Recursos (<i>Uniform Resource Identifier</i>)
USB	Barramento Serial Universal (<i>Universal Serial Bus</i>)

Lista de símbolos

D_F	Dimensão espacial do mapa de características
D_K	Dimensão espacial do <i>kernel</i> convolucional
h_k^a	Altura da caixa padrão
L_{Conf}	Perda de Confiança (<i>Confidence Loss</i>)
L_{Loc}	Perda de Localização (<i>Localization Loss</i>)
$L(x,c,l,g)$	Perda Geral
s_k	Escala das caixas padrões
s_{max}	Camada mais alta da caixa padrão
s_{min}	Camada mais baixa da caixa padrão
w_k^a	Largura da caixa padrão

Sumário

1	INTRODUÇÃO	15
1.1	Tema do Projeto	15
1.2	Problematização	16
1.3	Hipóteses	16
1.4	Objetivos	17
1.4.1	Objetivos Gerais	17
1.4.2	Objetivos Específicos	17
1.5	Justificativas	18
1.6	Considerações Finais	18
2	QR CODE	19
2.1	Estrutura do QR code	19
2.1.1	Padrões de detecção	20
2.1.2	Separadores	20
2.1.3	Padrão de temporização	21
2.1.4	Padrão de alinhamento	22
2.1.5	Formato da informação	23
2.1.6	Informação da versão	23
2.1.7	Zona silenciosa	24
2.2	Versões do QR code	24
2.3	Classificação do código QR code	25
2.3.1	Código estático	25
2.3.2	Código dinâmico	27
2.4	Estrutura de dados	28
2.5	Nível de correção de erro	31
2.6	Leitura do QR code	34
2.7	Considerações Finais	36
3	WECHAT QR CODE DETECTOR	38
3.1	Single Shot MultiBox Detector (SSD)	39
3.1.1	Intersection Over Union (IoU)	40

3.1.2	Método de seleção de caixas delimitadoras	41
3.1.3	Processo de treinamento	41
3.1.3.1	Estratégia de correspondência	42
3.1.3.2	Função de Perda Geral	42
3.1.3.3	Escalas e proporções das caixas padrões	44
3.1.3.4	Técnica <i>Hard Negative Mining</i>	44
3.1.3.5	Aumento de dados	45
3.2	<i>MobileNetV2</i>	45
3.2.1	<i>Depthwise Separable Convolution (DSC)</i>	45
3.2.2	<i>Linear Bottlenecks</i> (Blocos de Gargalo)	47
3.2.3	<i>Inverted Residuals</i> (Resíduos Invertidos)	48
3.2.4	Arquitetura	49
3.3	<i>Quick Response Super Resolution (QRSR)</i>	51
3.3.1	Concatenação <i>DenseNet</i>	51
3.3.1.1	Taxa de crescimento	52
3.3.1.2	Reutilização de recursos	52
3.3.2	Deconvolução	53
3.3.2.1	Extração e recuperação de <i>patches</i>	53
3.3.2.2	Mapeamento não linear	53
3.3.2.3	Reconstrução	54
3.4	Considerações finais	54
4	MATERIAIS E METÓDOS	55
4.1	Método	55
4.1.1	Aplicativo	55
4.1.2	Experimento	57
4.1.2.1	Decodificação e detecção	57
4.1.2.2	Codificação	58
4.2	Material	59
4.2.1	Aplicativo	59
4.2.2	Experimento	60
4.3	Considerações Finais	62
5	RESULTADOS E DISCUSSÕES	63
5.1	Resultados do experimento	63

5.1.1	Decodificação e detecção	63
5.1.1.1	Altura e iluminação	63
5.1.1.2	Rotação	67
5.1.2	Codificação	71
5.2	Considerações finais	73
6	CONCLUSÃO	75
	REFERÊNCIAS	78
7	APÊNDICE A - PORÇÃO DO CÓDIGO EM PYTHON PARA RE- ALIZAR A DECODIFICAÇÃO E CODIFICAÇÃO DE <i>QR CODES</i> .	81

1 Introdução

No período Pós-Segunda Guerra Mundial, a economia japonesa registrou um crescimento acelerado a uma taxa anual média de aproximadamente 10%, sendo duas vezes maior que a dos países desenvolvidos (1). Além da economia, o país expandiu abruptamente seu setor tecnológico ao introduzir um novo sistema de trabalho conhecido como Toyotismo. Com este novo sistema, a sociedade necessitava de uma forma eficiente de rastrear itens devido à produção em massa de produtos e ao aumento de grandes redes de supermercados (2). Devido às capacidades limitadas de armazenamento de informação do código de barras, criou-se um código, o *Quick Response Code (QR code)*, que poderia ser lido de várias direções, ângulos e distâncias (2).

O *QR code* é semelhante ao código de barras, que pode ser escaneado por dispositivos digitais como os *smartphones*. O *QR code* armazena a informação ao longo de uma série de pixels em preto e branco, distribuída em uma matriz quadrada (3). Diversas aplicações como aplicativos bancários, cardápios digitais e aplicativos de rastreabilidade fazem uso do código a fim de armazenar dados, codificar informações e/ou detectar objetos em trânsito (2).

O código foi desenvolvido na década de 90 pela Denso Wave, empresa subsidiária da Toyota, com o objetivo de rastrear os automóveis por meio do processo de manufatura. Com o seu grande sucesso, o *QR code* é altamente utilizado nas campanhas publicitárias para compartilhar informações aos usuários (3). Diferente do código de barras, que é escaneado na horizontal por *scanners* ópticos de luz, o *QR code* é escaneado tanto na horizontal quanto na vertical pelos aparelhos celulares modernos, por meio da câmera do dispositivo, que extrai os padrões da matriz (3).

A leitura dos dados, por meio de uma imagem matricial com padrões bem definidos, é mais desafiadora do que parece. Diversos fatores como altura, iluminação ambiente e imagem rotacionada acabam comprometendo a correta interpretação das informações confinadas na matriz.

1.1 Tema do Projeto

O tema deste projeto está relacionado a estrutura, aos procedimentos e desafios envolvidos na codificação e decodificação do *QR code*. Atualmente, o *QR code* é utili-

zado para trocar informações (como *links*, textos, pagamentos, contratos, *wi-fi*) entre os usuários, rastrear objetos, identificar items e documentar informações. O *QR code* esta disponível em quarenta versões, variando de 1 a 40.

Neste trabalho, foca-se no desenvolvimento de um aplicativo para o sistema *android* que facilite a leitura de códigos *QR code* de diferentes modelos e versões. Além disso, o trabalho busca também difundir conhecimento a respeito da estrutura, da codificação e decodificação do *QR code*, e também verificar as limitações do detector do aplicativo.

1.2 Problematização

Atualmente, a maioria dos indivíduos sabem como obter ou inserir informações do *QR code* por meio de aplicativos e *sites*, porém desconhecem a estrutura e os processos que envolvem a leitura e a escrita da informação contida no código. Embora os veículos de comunicação publiquem matérias e artigos sobre o funcionamento do código, eles não mostram em detalhes os processos necessários para codificar e decodificar o código ao público. Além disso, muitos artigos acadêmicos que estudam detalhadamente tais processos, geralmente, são inacessíveis ao público por serem pagos, restritos ou apenas estudantes universitários têm acesso.

Diversos aplicativos são capazes de acessar a câmera do celular para identificar e interpretar os códigos *QR code*. Através da câmera do *smartphone*, o aplicativo consegue capturar a imagem do código, transformando-a em um vetor para alimentar o algoritmo responsável pela interpretação do *QR code*. Entretanto, existem alguns fatores que interferem na interpretação dos dados. Estes fatores incluem a elevada quantidade de informação, iluminação ambiente, distância entre a câmera e o *QR code*, e rotação do código.

A partir dos questionamentos acima, foi possível nortear o desenvolvimento do projeto, e estabelecer os objetivos a serem alcançados. Dessa forma, é necessário revisar a teoria relacionada ao *QR code*, assim como a implementação do aplicativo para a leitura do código.

1.3 Hipóteses

No presente trabalho, ao desenvolver um aplicativo leitor de *QR code* e difundir o conhecimento a cerca do código, espera-se:

- Que o leitor consiga entender os conceitos de codificação e decodificação do *QR code*;
- Que o leitor conheça a estrutura, as versões e os modelos do *QR code*;
- Que o aplicativo consiga interpretar corretamente diferentes modelos e versões do *QR code*;
- A construção de um aplicativo utilizando as linguagens *Python* e *JavaScript*;
- Identificar as limitações do modelo de decodificação e detecção de *QR codes* conhecido como *WeChat*.

1.4 Objetivos

1.4.1 Objetivos Gerais

Além de difundir o conhecimento a respeito do *QR code*, o objetivo do projeto é melhorar ou adaptar os métodos utilizados para codificar e decodificar a informação do *QR code*, de diversos modelos e versões, através do desenvolvimento de um aplicativo dedicado à codificação, decodificação e identificação de códigos *QR code*.

1.4.2 Objetivos Específicos

- Revisar a teoria acerca do *QR code*;
- Revisar a teoria de redes neurais;
- Revisar conceitos básicos sobre lógica de programação;
- Revisar conceitos de desenvolvimento de aplicativos *mobile* no *android*;
- Testar o aplicativo para verificar sua funcionalidade em diferentes cenários de distância, iluminação e rotação;
- Testar o aplicativo utilizando *QR codes* de diferentes versões e níveis de correção de erro;
- Demonstrar a codificação de diferentes tipos de dados.

1.5 Justificativas

Durante a década de 2010, o *QR code* teve um crescimento constante, porém teve um crescimento exponencial durante a pandemia de COVID-19. Mesmo após a pandemia, a sociedade começou a buscar maneiras simples e práticas para obter quantidades volumosas de informações (2). Assim, conforme a tecnologia amadurece e o volume de dados cresce, é necessário desenvolver novas estratégias para lidar com quantidades volumosas de dados gerados diariamente na sociedade contemporânea. A criação deste aplicativo pode se transformar em uma ferramenta prática e de fácil uso, possibilitando a obtenção de informações contidas em códigos *QR code*.

No livro *Discurso do Método*, René Descartes diz que "para viver de maneira melhor e mais sabiamente, é necessário aprender e transmitir o conhecimento." (4). Ou seja, o filósofo reforça que a transmissão do conhecimento é importante para o progresso e desenvolvimento humano. Neste sentido, o projeto visa também disseminar o conhecimento teórico acerca do *QR code*, de forma a auxiliar no progresso tecnológico e social.

1.6 Considerações Finais

Devido à importância do *QR code* na divulgação de campanhas publicitárias e no compartilhamento de informações, o aplicativo a ser implementado neste trabalho pode ajudar o usuário em obter informações de códigos *QR code* de diversos modelos e versões.

Durante o desenvolvimento do projeto, é necessário consultar diversas referências relacionadas a desenvolvimento de aplicativos, estrutura do *QR code*, métodos de codificação e decodificação de *QR codes*, e ao modelo de AI a ser utilizado no projeto para decodificar e detectar os códigos. Assim, espera-se alcançar os objetivos do trabalho ao construir uma ferramenta que auxilie no desenvolvimento tecnológico e social.

2 QR code

O *QR code* consiste em vários módulos quadrados, denominados células, agrupados em uma matriz quadrada sobre um fundo branco (5). Devido a essa disposição, é possível armazenar mais conteúdo que um código de barras, ao inserir dados tanto na vertical quanto na horizontal, de forma que o dispositivo móvel possa identificá-lo em qualquer direção a 360° (6, 7).

Devido à alta capacidade de armazenamento, o *QR code* pode armazenar 7089 caracteres numéricos, 4296 caracteres alfanuméricos e 2953 *bytes* binários, além de 1817 caracteres chineses (6). Para gravar a informação e decifrá-la, são necessários um codificador e um decodificador. O codificador é encarregado de codificar o conteúdo recebido e gerar o *QR code*, enquanto o decodificador decifra a informação contida no *QR code* gerado (5). É possível visualizar o funcionamento do *QR code* na Figura 2.1.

Figura 2.1 – Diagrama de blocos do funcionamento do *QR code*.



Fonte: adaptado de (5).

Normalmente, as informações fornecidas ao codificador são textos, *links*, contatos e menus digitais, resultando no código requerido com base na informação recebida. Os dados são convertidos em uma sequência de números binários, formando um fluxo de *bits* (*bit-stream*), que é representado no *QR code*, e que por sua vez, é decifrado pelo decodificador (8).

2.1 Estrutura do *QR code*

Cada símbolo do *QR code* é formado pelo arranjo de diversos módulos quadrados segundo padrões de função e regiões de codificação (5). O símbolo precisa estar envolvido por uma margem, conhecida como zona silenciosa, localizada nos quatro lados da matriz (8).

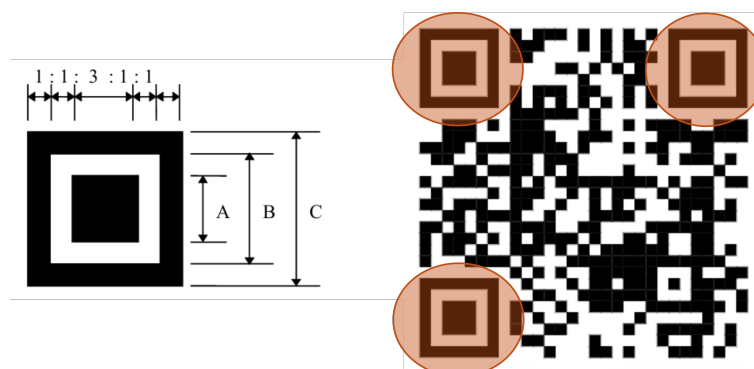
A inserção de padrões de função em áreas específicas é crucial para assegurar a correta interpretação da informação pelos leitores de código *QR code*. Os leitores precisam identificar e direcionar adequadamente o código para o processo de decodificação (5). Há quatro categorias de padrões de função: o padrão de detecção, o separador, o padrão de temporização e o padrão de alinhamento (8).

Os dados codificados são introduzidos na região de codificação que contém a informação, o formato da mesma e a correção de erros de palavras-código (5). Os padrões de função não devem ser empregados para codificar os dados, mas sim para facilitar a detecção do *QR code* pela câmera do celular (9).

2.1.1 Padrões de detecção

Os padrões de detecção são designados para direcionar adequadamente o *QR code* durante o processo de leitura e decodificação (5). Esses padrões apresentam três estruturas situadas nas extremidades do *QR code*, exceto no canto inferior direito (8). Todas possuem uma matriz 7x7 de células escuras, uma matriz 5x5 de células claras, no interior da primeira matriz, e uma matriz 3x3 de células escuras localizada no centro (5).

Figura 2.2 – Padrão de detecção.



Fonte: adaptado de (5).

A proporção de células escuras e claras nas três matrizes é de 1:1:3:1:1. Tal relação aprimora a eficácia de leitura do *QR code* em qualquer direção (10).

2.1.2 Separadores

Os separadores são espaços vazios de um módulo de largura entre os padrões de detecção (5), utilizados para distingui-los da informação (11).

Figura 2.3 – Separador.



Fonte: adaptado de (5).

2.1.3 Padrão de temporização

O padrão de temporização é essencial na determinação da densidade do símbolo, das coordenadas dos módulos e da área que contém as informações sobre a versão do *QR code* (5). Trata-se de uma faixa alternada de módulos claros e escuros que se inicia no canto inferior esquerdo do padrão de detecção localizado no canto superior direito, indo para o padrão de detecção do canto superior esquerdo até o padrão de detecção do canto inferior esquerdo (9). Há dois padrões de temporização: o horizontal e o vertical. O horizontal está localizado na sexta linha, enquanto o vertical está localizado na sexta coluna do *QR code* (5).

Figura 2.4 – Padrão de temporização.



Fonte: adaptado de (9).

2.1.4 Padrão de alinhamento

O padrão de alinhamento é responsável por corrigir a distorção do *QR code* (6). Isso permite que o detector reconheça uma imagem distorcida e a transforme em uma grade de módulos claros e escuros da informação codificada (9). A configuração do padrão de alinhamento consiste em uma matriz 5x5 de módulos escuros, uma matriz 3x3 de módulos claros e um módulo escuro no centro (5). O módulo central tem a função de identificar a distorção do *QR code* (6).

Figura 2.5 – Padrão de alinhamento.



Fonte: adaptado de (5).

A localização do padrão de alinhamento varia de acordo com a quantidade de informações, o tamanho e a versão do código *QR code*. Além disso, quanto mais extenso for o *QR code*, mais padrões de alinhamento serão requeridos (12). Na Tabela 2.1, é possível observar a localização dos padrões de alinhamento de algumas versões.

Tabela 2.1 – Tabela de coordenadas dos padrões de alinhamento

Versão	Linha e coluna		Versão	Linha e coluna	
2	6	18	15	6	26 48 70
3	6	22	20	6	30 60 90
4	6	26	25	6	30 58 82 110
5	6	30	30	6	30 62 94 130
10	6	28 50	40	6	30 58 86 114 142 170

Fonte: adaptado de (13).

Os números da coluna à direita representam as coordenadas de linha e coluna. A versão 1 não possui tal elemento, o que explica a sua ausência na tabela.

A versão 40 possui 46 padrões de alinhamento, alguns dos quais foram eliminados por causa da sobreposição com os padrões de detecção ou separadores (13).

2.1.5 Formato da informação

O formato da informação está localizado ao redor dos três padrões de detecção. O formato da informação inclui dados essenciais sobre o grau de tolerância a erros e os padrões de máscara do *QR code* (12). Ao codificar a informação, alguns módulos são invertidos, seguindo uma regra predefinida, com o intuito de aprimorar a legibilidade e prevenir a formação de aglomerados de módulos da mesma cor. Este procedimento é denominado mascaramento (*masking*) e serve para avisar ao decodificador que certos módulos foram invertidos. O formato da informação é codificado em aproximadamente 15 *bits*, sendo que um dos formatos da informação, situado ao redor do padrão de detecção no canto superior esquerdo, tem 15 *bits*. Os outros dois possuem 7 *bits* e 8 *bits* (9).

Figura 2.6 – Localização do formato a informação.



Fonte: adaptado de (5).

2.1.6 Informação da versão

O *QR code* possui 40 variações, cada uma com diferentes quantidades de módulos (12). Esses dados são codificados em duas matrizes 6x3, situadas perto dos padrões de detecção, nos cantos superior direito e no inferior esquerdo (9).

Figura 2.7 – Informação da versão.



Fonte: adaptado de (5).

2.1.7 Zona silenciosa

A zona silenciosa, também chamada de margem, é um espaço amplo de 4 módulos, projetado para prevenir o desvio de informações de textos ou marcações presentes no *QR code* (5).

Figura 2.8 – Zona silenciosa.



Fonte: adaptado de (12).

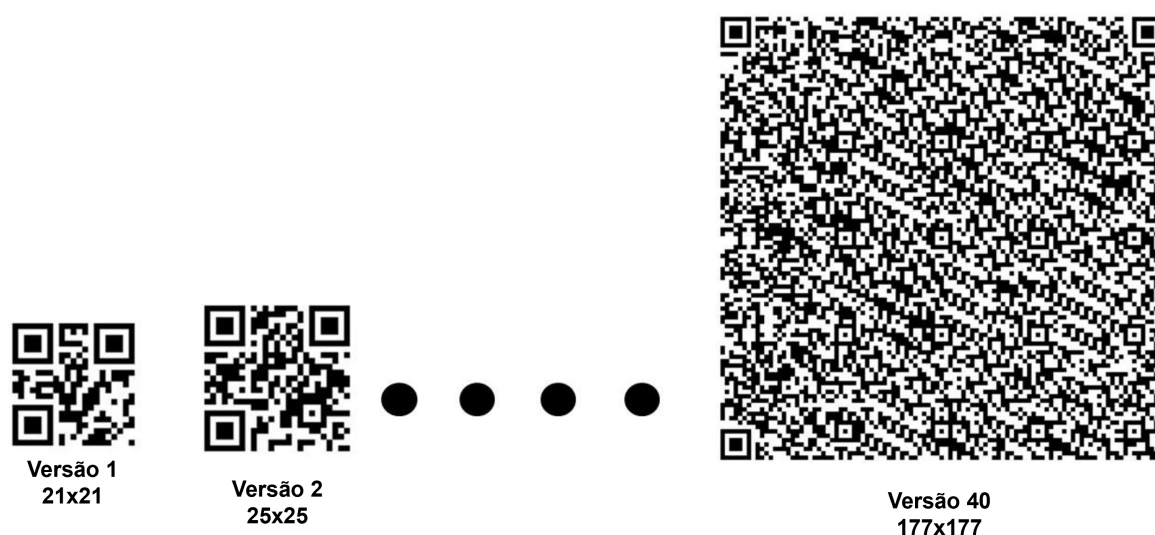
2.2 Versões do *QR code*

As versões do *QR code* oscilam entre a versão 1 e a 40. Cada versão estabelece uma configuração específica para a dimensão do código, determinando o número exato de módulos que compõem a sua matriz geométrica (5).

Cada versão possui um limite máximo de armazenamento de informações, que varia conforme a quantidade de informação inserida, o tipo de caractere e o grau de correção de erros (5). À medida que o volume de dados cresce, torna-se imprescindível

adicionar mais módulos, ou seja, ampliar as dimensões do *QR code*. Por exemplo, a versão 1 representa uma matriz de 21x21, a versão 2, uma matriz de 25x25, a versão 3, uma matriz de 29x29, a versão 10, uma matriz de 57x57, a versão 25, uma matriz de 117x117, e a versão 40, uma matriz de 177x177 (14).

Figura 2.9 – Diferença entre as versões do *QR code*.



Fonte: adaptado de (5).

2.3 Classificação do código *QR code*

O *QR code* é classificado em duas categorias principais: código estático e código dinâmico. O *QR code* estático é permanente, isto é, uma vez gerado, não pode ser modificado novamente, ao passo que o dinâmico permite atualizar a informação sem a necessidade de gerar um novo código (14).

2.3.1 Código estático

O código estático não fornece a habilidade de rastrear ou interpretar dados de digitalização, mas sim de armazenar a informação. As informações armazenadas não são constantemente atualizadas, ou seja, uma vez inseridas, elas não podem ser modificadas no próprio QR code, a menos que seja gerado um novo. Dados como números telefônicos, e-mails, números seriais e textos são comumente incluídos nos códigos estáticos (14).

As primeiras versões da Denso Wave, os modelos 1 e 2 do padrão ISO/IEC 18004:2000, foram lançadas no ano 2000 respectivamente, com o modelo 2 se destacando

por apresentar padrões de alinhamento e maior capacidade de armazenamento (14).

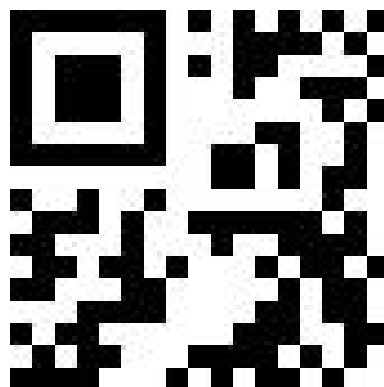
Figura 2.10 – *QR code* nos modelos 1 e 2.



Fonte: adaptado de (14).

Adicionalmente, foi disponibilizada a versão micro do *QR code*, uma versão reduzida em relação aos dois modelos anteriores, ideal para aplicações menores que requerem dimensões menores (14).

Figura 2.11 – Micro *QR code*.



Fonte: adaptado de (5).

Há quatro versões que vão de M1 a M4, sendo a M4 a que armazena mais dados (15). A Tabela 2.2 mostra um comparativo entre as versões do micro *QR code* em termos de capacidade de armazenamento, considerando o nível de correção de erro descrito na seção 2.5.

Tabela 2.2 – Tabela comparativa da capacidade de armazenamento entre as versões do Micro *QR code*

Versão	Num. módulos	Nível correção de erro	Numérico	Alfanumérico	Binário
M1	11	-	5	-	-
M2	13	L	10	6	-
		M	8	5	-
M3	15	L	23	14	9
		M	18	11	7
M4	17	L	35	21	15
		M	30	18	13
		Q	21	13	9

Fonte: (15)

Por fim, a versão *frame* do *QR code* é um símbolo que possui uma área central que possibilita a inserção de elementos gráficos e textuais sem alterar a informação. O código possibilita a customização de diversas formas e cores, tornando várias aplicações versáteis (14).

Figura 2.12 – Versão *frame*.



Fonte: autor.

2.3.2 Código dinâmico

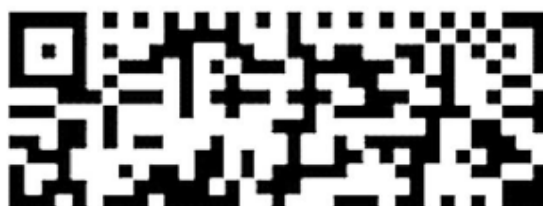
Os códigos dinâmicos são mais sofisticados que os estáticos, pois incorporam uma URL de redirecionamento (15). Os leitores são direcionados aos dados incorporados em cada código, que podem ser atualizados sem a necessidade de gerar um novo (14).

O uso da URL garante que o tamanho da informação não interfira na contagem de módulos. Além disso, os códigos dinâmicos fornecem detalhes adicionais como a quan-

tidade e a localização dos leitores, além de serem personalizados e efetivos em aplicações de registros (14).

Uma das primeiras versões, desenvolvida pela Denso Wave, foi o *iQR code*. Trata-se de uma matriz bidimensional, de fácil leitura, que possibilita o armazenamento de mais informação em comparação aos códigos estáticos, utilizando 30% de espaço a menos (14). O *iQR code* foi designado a situações em que um formato mais longo e estreito é mais prático quando o leitor está mais distante.

Figura 2.13 – *iQR code*.



Fonte: adaptado de (5).

Outro modelo desenvolvido foi o *Secure Quick Response Code* (SQRC), destinado ao armazenamento seguro de dados confidenciais. O SQRC é capaz de salvar informações tanto privadas quanto públicas. A informação privada pode ser acessada por um leitor dedicado que possua a chave privada (16).

2.4 Estrutura de dados

A informação armazenada engloba tanto as informações codificadas, fornecidas pelo usuário, quanto a correção de erro de *bits*. O algoritmo *Reed Solomon* faz parte da estrutura do *QR code*, com o objetivo de incorporar o detector e o corretor de erros na fonte de dados (9).

Há três tipos fundamentais de dados: numéricos, alfanuméricos e binários. Além dos três, existem ainda o kanji e o ECI (13). Cada versão do *QR code* possui um limite máximo de armazenamento de dados, e à medida que o volume de dados cresce, mais módulos são necessários para guardar toda a informação (9).

Logo abaixo, a Tabela 2.3 mostra a capacidade de armazenamento de dados numéricos e alfanuméricos em algumas versões. De acordo com a tabela, a quantidade de informação a ser armazenada varia conforme o nível de correção de erro. Observe que, para todas as versões, o nível H tem a menor capacidade de armazenamento.

Tabela 2.3 – Capacidade de dados do tipo numérico e alfanumérico por nível de correção de erro

Versão	Módulos	Nível de correção de erro	Numérico	Alfanumérico
1	21x21	L	41	25
		M	34	20
		Q	27	16
		H	17	10
2	25x25	L	77	47
		M	63	38
		Q	48	29
		H	34	20
3	29x29	L	127	77
		M	101	61
		Q	77	47
		H	58	35
4	33x33	L	187	114
		M	149	90
		Q	111	67
		H	82	50
14	73x73	L	1,101	667
		M	871	528
		Q	621	383
		H	468	283
40	177x177	L	7,089	4,296
		M	5,596	3,391
		Q	3,993	2,420
		H	3,057	1,852

Fonte: (9)

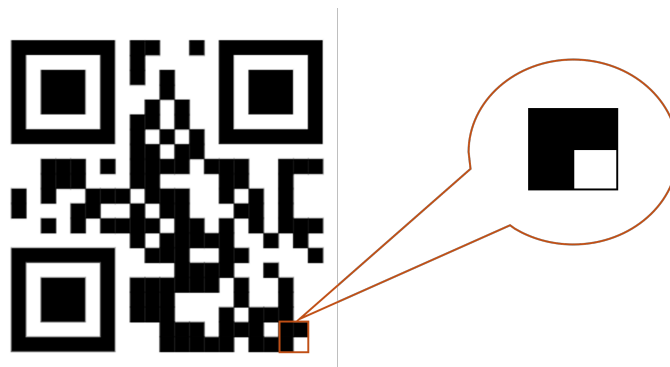
Já a Tabela 2.4 mostra a representação binária dos cinco tipos de dados apresentados.

Tabela 2.4 – Representação binária dos tipos de dados

Tipo	Bits	Pixels
Numérico	0001	
Alfanumérico	0010	
Byte	0100	
Kanji	1000	
ECI	0111	

Fonte: (13)

Os quatro *bits* são posicionados no canto inferior direito, conforme é visto na Figura 2.14 (13).

Figura 2.14 – Posição dos *bits* de tipo de dado

Fonte: adaptado de (13).

De acordo com a ISO/IEC 18004, os modos de classificação contêm os seguintes caracteres:

- Dígitos decimais de 0 a 9;
- Alfanuméricos letras, caracteres especiais e espaços.

As codificações alfanuméricas e numéricas podem ser analisadas na Tabela 2.5.

Tabela 2.5 – Codificações alfanuméricas

Caractere	Codificação	Caractere	Codificação	Caractere	Codificação
0	0	F	15	U	30
1	1	G	16	V	31
2	2	H	17	W	32
3	3	I	18	X	33
4	4	J	19	Y	34
5	5	K	20	Z	35
6	6	L	21	Espaço	36
7	7	M	22	\$	37
8	8	N	23	%	38
9	9	O	24	*	39
A	10	P	25	+	40
B	11	Q	26	-	41
C	12	R	27	.	42
D	13	S	28	/	43
E	14	T	29	:	44

Fonte: (13)

Conforme a ISO/IEC 18004, caso a informação codificada consistir apenas em dígitos numéricos, o modo numérico será aplicado. No entanto, se o modo numérico não for suficiente para o preenchimento da informação, então o modo alfanumérico será acionado. Para caracteres ausentes na Tabela 2.5, usamos a tabela ASCII como referência (13).

A codificação binária é empregada pois os leitores são capazes de reconhecer os padrões da imagem em sequência binária. Os *bytes* têm a função de carregar a mensagem contida do *QR code*. Os *bytes* são compostos por 8 *bits*. O número de *bits* pode variar de acordo com a versão (13).

2.5 Nível de correção de erro

Em certas circunstâncias, o *QR code* pode ser danificado fisicamente ou ser corrompido de forma a comprometer a sua decodificação. Em caso de dano, os dados podem ser interpretados em até 30% (9). Uma alternativa seria a implementação do nível de correção de erro (ECC), que reserva uma porção do código para a reiteração dos dados da imagem (13).

Os dois elementos de correção de erros estão localizados perto dos separadores do canto superior e inferior esquerdo, conforme é ilustrado na Figura 2.15

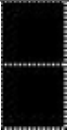
Figura 2.15 – Elementos de correção de erro



Fonte: adaptado de (13).

O *Low Level* (L), *Medium Level* (M), *Quality Level* (Q) e *High Level* (H) são os quatro níveis de correção de erro. Cada nível adiciona quantidades variadas de informações de *backup*, dependendo do grau de dano que o *QR code* suporta (9). Conforme apresentado na Tabela 2.6, é possível visualizar o percentual de restauração e a representação binária dos quatro níveis de correção de erro.

Tabela 2.6 – Níveis de correção de erro

Nível corretor	Bits	Percentual de restauração	Pixels
L	01	7%	
M	00	15%	
Q	11	25%	
H	10	30%	

Fonte: adaptado de (9, 13)

O fluxo de dados, gerado pelo codificador, é dividido em uma sequência de palavras-código de 8 *bits*. Ao utilizar níveis de correção de erros elevados, como o nível H, a capacidade de recuperação das palavras-código aumenta de forma significativa. Todavia, a quantidade de informação a ser de codificada também aumenta (17).

Para cada bloco, as palavras-código de correção de erros são representados por (c,k,r) , onde c representa a quantidade total de palavras-código, k é a quantidade de dados da palavra-código, e r indica a capacidade de correção de erros. Neste cenário, a capacidade de correção de erros corresponde ao número máximo de palavras-código que são modificadas por bloco (17).

Dado um *QR code* com nível de correção do tipo H, o número de *bits* de uma mensagem embutida é determinado pela capacidade de correção de erro do código (17).

$$n = \left\lfloor \frac{m - e}{2} \right\rfloor \times 8 \leq \left\lfloor \frac{m}{2} \right\rfloor \times 8 \quad (2.1)$$

Na Equação 2.1, n é o número de *bits* da mensagem, m indica o número de palavras-código de correção de erros, e e representa o número de palavras-código de detecção de erros. A quantidade máxima de palavras-código alteradas no *QR code* é definida como l , sendo $l = n/8$ (17).

A quantidade total de palavras-código para cada versão do *QR code* é definida como N , onde $N = m + t$, sendo t o número total de palavras-código de dados (17). A partir de N , é possível determinar a taxa de correção de erros para o tipo H.

$$H = \frac{m}{2N} = \frac{m}{2(m + t)} \quad (2.2)$$

A relação entre a quantidade total de palavras-código de correção de erro e de palavras-código de dados é descrita na Equação 2.3.

$$\frac{m}{t} = \frac{2H}{1 - 2H} \quad (2.3)$$

O valor m/t oscila entre 1,7 e 1,96, de acordo com as pesquisas realizadas com as versões 2 a 40 do *QR code*. A partir desta relação, a taxa de correção de erro do tipo H varia de 31,5% a 33,2%. Em alguns casos, a taxa pode ser maior que 33,2% ou menor que 31,5% conforme o método da região for alterado (17).

2.6 Leitura do *QR code*

Dentro de cada *byte*, os *bits* são alocados de forma a respeitar a ordem mostrada na Figura 2.16.

Figura 2.16 – Distribuição dos *bytes*



(a) Distribuição de baixo para cima (b) Distribuição de cima para baixo

Fonte: (13)

Os *bits* serão distribuídos de baixo para cima ou de cima para baixo, conforme a sequência de alocação dos *bytes* (13).

Como é mostrado na Figura 2.17, o *byte* também pode ser distribuído na horizontal em uma matriz 2x4 (13).

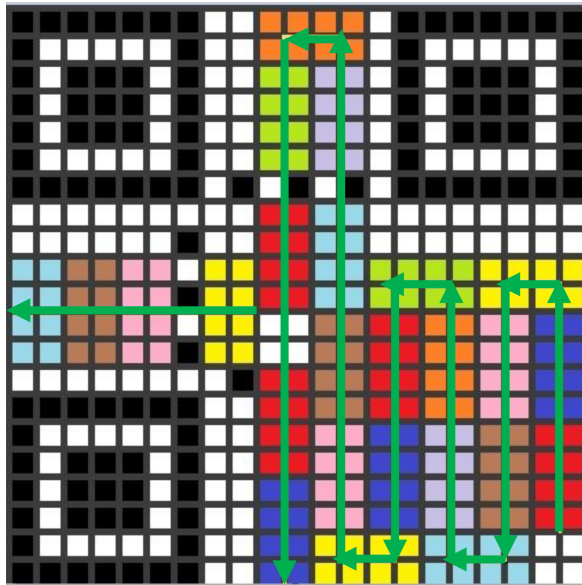
Figura 2.17 – Distribuição dos *bytes* na horizontal



Fonte: (13)

Cada *bit* possui um valor específico dentro dos *bytes*. O primeiro *bit* é o mais significativo, enquanto o último é o menos significativo.

Na mensagem, os *bytes* devem seguir o padrão representado na Figura 2.18. O primeiro *byte*, destacado em vermelho, deve conter o número total de *bytes* da mensagem. Ele deve ser posicionado acima dos quatro primeiros *bits* que contém a classificação dos dados (numéricos, alfanuméricos, binários, kanji e ECI) do *QR code* (13).

Figura 2.18 – Caminho dos *bits*

Fonte: adaptado de (13).

As setas em verde seguem a ordem do caminho da informação do QR code. A ordem dos bytes dentro do código pode ser visualizada na Tabela 2.7

Tabela 2.7 – Posições dos *bytes*

Posição	Cor
1°	Vermelho
2°	Azul Escuro
3°	Amarelo
4°	Rosa
5°	Marrom
6°	Azul Claro
7°	Cinza
8°	Laranja
9°	Verde
10°	Vermelho
11°	Azul Escuro
12°	Amarelo
13°	Rosa
14°	Marrom
15°	Azul Claro
16°	Cinza
17°	Laranja
18°	Verde
19°	Vermelho
20°	Vermelho
21°	Azul
22°	Amarelo
23°	Rosa
24°	Marrom
25°	Azul Claro

Fonte: (13)

Na Figura 2.18, é observável que existe um bloco de quatro *bits* em branco entre os *bytes* 19 e 20 da sequência. Este bloco indica o fim da mensagem original, sendo a mensagem repetida até preencher o espaço restante do *QR code*. A repetição é necessária para que o leitor consiga restaurar algum dado que esteja danificado (13).

2.7 Considerações Finais

O *QR code* é formado por diversos módulos quadrados, denominados células, que são agrupados em uma matriz quadrada em um fundo branco. Devido a essa característica, o *QR code* consegue armazenar mais informações do que o código de barras.

A estrutura do *QR code* pode ser dividida em padrões de função e regiões de codificação. Os padrões de função, classificados em padrão de detecção, separador, pa-

drão de temporização e padrão de alinhamento, são fundamentais para garantir a correta interpretação da informação pelos leitores de código *QR code*. Por outro lado, as regiões de codificação contêm a informação, o formato e o nível de correção de erro da mensagem.

O *QR code* é classificado em duas categorias principais: código estático e código dinâmico. O código estático é permanente e não pode ser modificado, ao passo que o código dinâmico pode ser atualizado sem a necessidade de criar um código.

Por fim, as informações fornecidas pelo usuário, bem como as informações codificadas, devem ser incluídas. Os dados armazenados são categorizados em numéricos, alfanuméricos, binários, kanji e ECI.

3 *WeChat QR Code Detector*

No campo da inteligência artificial para detecção de objetos e visão computacional, o *OpenCV* é conhecido por ser uma solução amplamente utilizada, com mais de 40 milhões de *downloads* mensalmente, de acordo com a página do projeto (18). O *OpenCV* é uma biblioteca de código aberto que contém mais de 2500 algoritmos otimizados de *machine learning* e visão computacional. Esses algoritmos são bastante utilizados para detecção e reconhecimento facial, identificar objetos, rastrear movimentos da câmera, produzir imagens de alta resolução e dentre outros (18).

Entre os 2500 algoritmos, há um detector de *QR code* de alto desempenho amplamente empregado em vários aplicativos da Tencent (19), como as redes sociais chinesas *WeChat* e *WeCom* (19). Esse detector é denominado *WeChat QR Code Detector*, um detector e decodificador de alto desempenho, baseado em CNN, integrado aos módulos de contribuição do *OpenCV* conhecido como *opencv_contrib*. O detector apresenta quatro características principais:

- Detector de *QR code* fundamentado em *Convolutional Neural Network (CNN)*. O *WeChat* conta com um modelo de CNN compacto capaz de identificar múltiplos códigos. O detector utiliza a arquitetura *Single Shot MultiBox Detector (SSD)* com uma estrutura de *backbone*. A estrutura *backbone* possui semelhanças com a estrutura *backbone MobileNetV2*, entretanto o *backbone* é executado no *framework* de inferência *Caffe* (19).
- Código aprimorado fundamentado em CNN. Foi desenvolvido um modelo de CNN de super-resolução mais leve para melhorar o desempenho de *QR codes* pequenos, chamado de *Quick Response Super Resolution (QRSR)*. As principais técnicas utilizadas do modelo QRSR são a convolução em profundidade, concatenação DenseNet e deconvolução (19).
- Identificação de padrões de detecção mais sólida. Além da busca convencional em linhas horizontais, propõe-se um método de identificação de padrões baseado na área. O padrão de detecção é identificado ao calcular a área do bloco preto e branco, utilizando as células conectadas pré-computadas (19).

- Otimização massiva. Com base no *zxing-cpp* (20), biblioteca para leitura de códigos de barra, foi realizada uma otimização massiva com o intuito de aumentar a taxa de sucesso da decodificação do código. Foram testados mais métodos de binarização, para suportar a detecção de padrões de N:1:3:1:1, procurar mais padrões de alinhamento e agrupar mais padrões de detecção de tamanhos semelhantes (19).

Neste capítulo, a estrutura dos dois modelos CNN robustos é descrita em detalhes: o modelo de detecção SDD, *MobileNetV2*, e o modelo de super-resolução QRSR.

3.1 *Single Shot MultiBox Detector (SSD)*

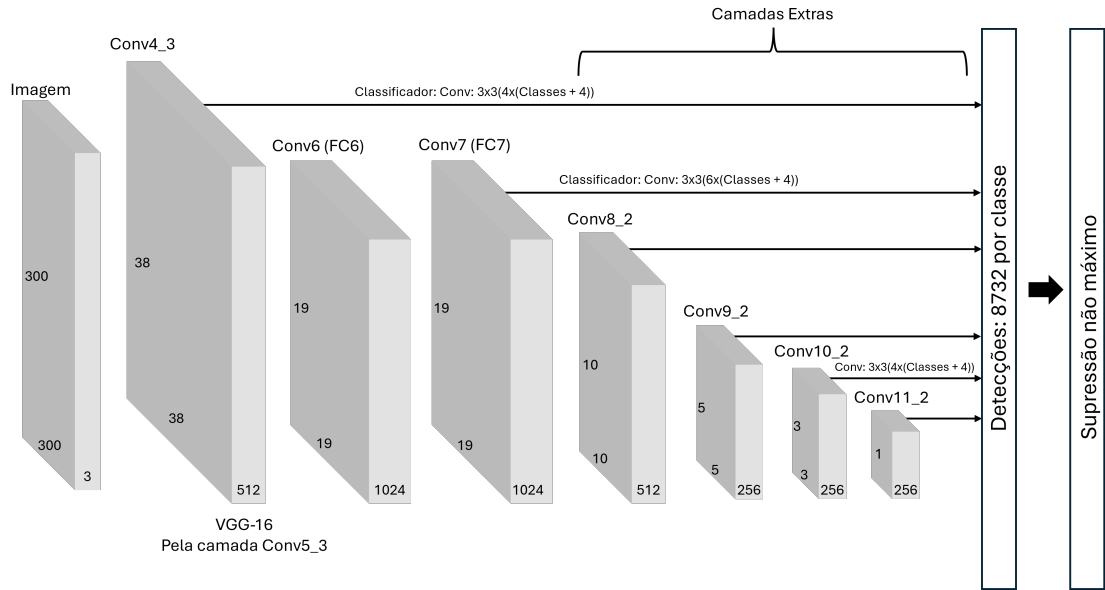
O método de detecção SSD fundamenta-se em uma rede convolucional *feed-forward*, que gera uma coleção de caixas delimitadoras de tamanho fixo e algumas pontuações para indicar a presença de instâncias da classe do objeto nessas caixas, seguida por uma etapa de supressão não máxima para gerar as detecções conclusivas (21).

As camadas neurais das redes anteriores seguem a arquitetura padrão para classificar imagens de alta qualidade. Antes das camadas de classificação, as camadas foram truncadas, levando a equipe responsável pela construção do modelo a adicionar uma estrutura auxiliar na rede. Essa estrutura foi projetada para gerar detecções que incluem mapas de características multiescala, preditores convolucionais para detecção, caixas padrão e proporções de aspecto (21).

Nos mapas multiescala, são inseridas camadas de recursos convolucionais ao final da rede truncada. Essas camadas diminuem progressivamente para previsões de detecções em múltiplas escalas. O modelo convolucional para previsão de detecções deve ser diferente para cada camada (21).

Cada camada inserida pode produzir um conjunto fixo de previsões de detecção utilizando um conjunto de filtros convolucionais. É possível visualizá-los na Figura 3.1. Para cada camada de tamanho $m \times n$ com p canais, existe um pequeno *kernel* de tamanho $3 \times 3 \times p$ para prever parâmetros de uma detecção em potencial. O *kernel* produz uma pontuação para uma categoria ou um deslocamento em relação às coordenadas da caixa padrão. É produzido um valor de saída em cada uma das $m \times n$ localizações. Os valores de saída do deslocamento da caixa delimitadora são calculados em relação a uma posição da caixa padrão de acordo com cada localização do mapa de características (21).

Figura 3.1 – Modelo de camadas SSD



Fonte: adaptado de (21).

O conjunto de caixas delimitadoras padrão é associado a cada célula do mapa de características, para múltiplos mapas de características no topo da rede. O preenchimento do mapa é feito de maneira convolucional, de forma que a localização de cada caixa em relação à sua célula correspondente permaneça inalterada. Em cada célula do mapa, são projetados deslocamentos em relação aos formatos das caixas padrão na célula, assim como as pontuações por classe que sinalizam a existência de uma instância de classe em cada uma dessas caixas (21).

Em suma, o SSD é um método muito bem aproveitado em aplicações de detecção de objetos em tempo real por entregar uma ótima precisão, mantendo um tempo de inferência mais rápido. O SSD ultrapassa um obstáculo importante: a técnica de processamento ser suficientemente rápida para acompanhar a taxa de quadros da câmera, enquanto muitos grandes modelos de detecção de objetos em tempo real ainda precisam aprimorar essa abordagem (22).

3.1.1 *Intersection Over Union (IoU)*

Trata-se da razão entre a interseção da área de duas caixas delimitadoras e a soma de suas áreas. O objetivo é calcular a quantidade de sobreposição presente entre a caixa delimitadora de previsão e a caixa delimitadora real (23). É possível observar a métrica do IoU através da Equação 3.1.

$$IoU = \frac{TP}{TP + FN + FP} \quad (3.1)$$

Na Equação 3.1, considerando que são positivos os *pixels* associados a padrões procurados, TP indica os *pixels* verdadeiros positivos, isto é, *pixels* pertencentes à *bbox* de predição e à real, FN refere-se aos falso positivos, isto é, aqueles presentes na *bbox* real, mas não na de predição, e FP corresponde aos falso positivos, presentes na *bbox* de predição, mas não na real. O valor da pontuação do IoU será alta caso haja muita sobreposição entre a *bbox* de predição e a *bbox* real. Caso contrário, o IoU terá uma pontuação mais baixa. Uma pontuação IoU igual a 1 indica uma correspondência perfeita entre o resultado previsto e o resultado real, enquanto uma pontuação nula indica que não houve sobreposição entre as duas *bboxes* (23).

A métrica IoU é bastante útil para analisar o quão bem um modelo possa identificar os itens de uma imagem.

3.1.2 Método de seleção de caixas delimitadoras

Conforme mencionado no início deste capítulo, o método SSD se fundamenta na rede convolucional *feed-forward*, complementado por uma fase de supressão não máxima para a obtenção de detecções. É possível reconhecer um conjunto de caixas delimitadoras (*bboxes*) como o mesmo objeto quando o valor IoU é maior que o valor limite (24). A supressão não máxima pode ser expressa a partir da Equação 3.2.

$$box = B_{\underset{i}{\operatorname{argmax}} C_i} \quad (3.2)$$

Na Equação 3.2, *box* indica o *bbox* com o maior valor de confiança, *B* representa o conjunto de *bboxes* e C_i corresponde ao valor de confiança da *i*-ésima *bbox*. Esses *bboxes* são selecionados como valores finais de saída. Entretanto, há uma possibilidade de que os resultados da detecção conttenham o valor máximo dos atributos. Portanto é inadequado ignorar os resultados da detecção de não-máximos sem uma análise prévia (24).

3.1.3 Processo de treinamento

A preparação dos dados e o treinamento da rede constituem os dois primeiros passos primordiais do processo de treinamento do SSD. O procedimento de coleta e armazenamento de um conjunto de dados de imagens tendo rótulos de objetos reais é denomi-

nado preparação dos dados. Durante a fase de treino da rede, há um ajuste no algoritmo SSD para prever com precisão o tipo e a posição dos objetos de uma imagem (22).

Um dos diferenciais de treinar o SSD é a necessidade de atribuir a informação real a saídas específicas no conjunto fixo de detectores de saída. Além disso, é necessário também a seleção do conjunto de caixas e escalas para a detecção, assim como elaborar estratégias de extração de *hard negative* e aumento de dados (21).

3.1.3.1 Estratégia de correspondência

Durante o treinamento do SSD, urge identificar quais caixas padrão estão associadas a uma detecção real fundamental e ajustar a rede de acordo com a situação. Para cada caixa real, é necessário selecionar entre caixas padrão que diferem na localização, proporção e escala. Diferentemente do *MultiBox*, posteriormente, as caixas padrão são combinadas com qualquer informação real que possua uma sobreposição de Jaccard superior a um limite de 0,5, simplificando o problema de aprendizado. Isso possibilita que a rede consiga atribuir altas pontuações da várias caixas padrões sobrepostas, ao invés de selecionar somente a que possui maior sobreposição como é visto na Equação 3.2 (21).

3.1.3.2 Função de Perda Geral

A função de otimização do treinamento é derivada do *MultiBox*, porém ela é estendida para trabalhar com múltiplas categorias de objetos. Seja $x_{ij}^p = \{1,0\}$ um indicador de correspondência da i -ésima caixa padrão com a j -ésima caixa padrão real da categoria p . Estrategicamente, é necessário que $\sum_i x_{ij}^p \geq 1$ (21).

A função de perda geral é a soma ponderada da *location loss* (L_{Loc}) e da *confidence loss* (L_{Conf}). O objetivo é penalizar o modelo tanto por errar a classe do objeto quanto por errar a posição da caixa (21). A Equação 3.3 descreve a perda geral.

$$L(x,c,l,g) = \frac{1}{N}(L_{conf}(x,c) + \alpha L_{loc}(x,l,g)) \quad (3.3)$$

Na Equação 3.3, N é o número de caixas padrões positivas, isto é, associadas aos padrões procurados de acordo com o valor de IoU. Para $N = 0$, a perda é definida como sendo nula. O parâmetro c representa as previsões de confiança (classes) feitas pela rede, l denota a localização da caixa prevista pela rede, g representa a caixa delimitadora real e α é o peso para balancear a importância entre acertar a localização e a classe. Geralmente, o valor de α é configurado para 1, indicando que a perda localizada é relevante nas análises (21).

A perda de localização é uma perda suavizada entre os parâmetros da caixa de previsão (l) e da caixa real (g). Os deslocamentos são regredidos para o centro (cx, cy) da *bbox* padrão (d) até a sua largura (w) e altura (h). Ela quantifica o erro do modelo ao ajustar as dimensões e a posição das caixas delimitadoras previstas em relação às caixas reais presentes (21). A Equação 3.4 descreve a perda de localização.

$$L_{loc}(x, l, g) = \sum_{i \in Pos}^N \sum_{m \in \{cx, cy, w, h\}} x_{ij}^k \text{smooth}_{L1}(l_i^m - \hat{g}_j^m) \quad (3.4)$$

$$\hat{g}_j^{cx} = (g_j^{cx} - d_i^{cx})/d_i^w \quad \hat{g}_j^{cy} = (g_j^{cy} - d_i^{cy})/d_i^h$$

$$\hat{g}_j^w = \log\left(\frac{g_j^w}{d_i^w}\right) \quad \hat{g}_j^h = \log\left(\frac{g_j^h}{d_i^h}\right)$$

A Equação 3.4 utiliza a função de erro smooth_{L1} para calcular a diferença entre os ajustes previstos pela rede (l) e os ajustes ideais ($\hat{g}$) para o centro da caixa (cx, cy), além da largura w e h das caixas. O somatório principal garante a aplicação da penalidade para às caixas positivas ($i \in Pos$), ignorando o fundo da imagem. As variáveis de ajuste ideal para largura e altura ($\hat{g}^w$ e $\hat{g}^h$) usufruem de operações logarítmicas para normalizar a escala, garantindo que erros de predição em objetos maiores ou menores sejam matematicamente proporcionais (21).

A perda de confiança é a perda *softmax* sobre múltiplas classes de confiança (c) do treino. Ela avalia a precisão do modelo em classificar corretamente o conteúdo de cada caixa delimitadora através da métrica de entropia cruzada (21). A Equação 3.5 descreve a perda de confiança.

$$L_{conf}(x, c) = - \sum_{i \in Pos}^N x_{ij}^p \log(\hat{c}_i^p) - \sum_{i \in Neg} \log(\hat{c}_i^0) \quad \text{onde} \quad \hat{c}_i^p = \frac{\exp(c_i^p)}{\sum_p \exp(c_i^p)} \quad (3.5)$$

A Equação 3.5 é dividida em dois somatórios distintos para lidar com a diferença de volume entre objetos reais e o cenário. O primeiro somatório atua sobre as caixas positivas ($i \in Pos$), a qual penaliza o modelo caso não atribua a probabilidade máxima à classe do objeto detectado. Já o segundo somatório atua sobre um subconjunto de caixas negativas ($i \in Neg$) que aplica uma penalidade quando a rede sinaliza falsamente a presença de um objeto em uma região que corresponde ao fundo da imagem capturada (representado pela classe 0 na equação). O termo $\hat{c}_i^p$ representa a aplicação da função *softmax* que converte as pontuações brutas da rede (c_i^p) em um espectro de probabilidades contínuas (21).

3.1.3.3 Escalas e proporções das caixas padrões

Uma abordagem para lidar com objetos de diversas escalas e proporções é processar a imagem em tamanhos variados e mesclar os resultados alcançados, a fim de ensinar o modelo a se ajustar a diferentes situações. É possível imitar o mesmo efeito ao utilizar mapas de características de várias camadas diversificadas em uma única previsão ao compartilhar os parâmetros em todas as escalas de objetos (21).

Considerando m a quantidade de mapas de características da predição, é possível computar a escala das caixas padrões de cada mapa de características utilizando a Equação 3.6.

$$s_k = s_{\min} + \frac{s_{\max} - s_{\min}}{m - 1}(k - 1), \quad k \in [1, m] \quad (3.6)$$

Na Equação 3.6, $s_{\min}$ representa a camada mais baixa e $s_{\max}$ a camada mais alta dentro do modelo. Todas as camadas intermediárias devem ser espaçadas regularmente. Além disso, é possível calcular a largura e a altura de cada caixa padrão pela Equação 3.7.

$$w_k^a = s_k \sqrt{a_r} \quad h_k^a = s_k / \sqrt{a_r} \quad (3.7)$$

Na Equação 3.7, a_r representa as proporções das caixas padrões, que por padrão são $a_r \in \left\{1, 2, 3, \frac{1}{2}, \frac{1}{3}\right\}$ (21).

Ao combinar as predições para todas as caixas padrões com diferentes escalas e proporções das localizações dos mapas de características, é realizável a construção de um conjunto diverso de predições, cobrindo diversos tamanhos e formas dos objetos de entrada (21).

3.1.3.4 Técnica *Hard Negative Mining*

Quando há um grande número de caixas padrão possíveis, a maioria delas apresenta resultados negativos após a etapa de correspondência, causando um desequilíbrio entre os resultados de treinamento positivo e negativo. Uma forma de contornar isso é classificar os resultados negativos utilizando a maior perda de confiança em cada caixa padrão e escolher os melhores para tornar a proporção entre negativos e positivos de no máximo 3:1, levando a uma otimização mais rápida e a um treino mais estável (21, 22).

3.1.3.5 Aumento de dados

Para tornar o modelo mais robusto a diferentes tamanhos e formatos de objetos de entrada, cada imagem de treinamento deve ser amostrada de forma aleatória. Pode-se amostrar as imagens usando a imagem de entrada original completa, escolhendo uma área em que a sobreposição mínima de Jaccard com os objetos seja de 0,1, 0,3, 0,5, 0,7 ou 0,9, ou selecionando uma área de forma aleatória (21).

A parte sobreposta da caixa contendo a informação real é mantida caso seu centro esteja na parte amostrada. Em seguida, cada parte amostrada é redimensionada para um tamanho fixo e invertida horizontalmente com probabilidade de 0,5. Além disso, são aplicadas algumas distorções fotométricas para treinar melhor o modelo (21, 22).

3.2 *MobileNetV2*

Além do SSD, o *WeChat* emprega uma estrutura de *backbone* fundamentada no modelo *MobileNetV2* (19). Howard analisou que o modelo *MobileNetV2* é bastante eficiente em dispositivos móveis, visão computacional e dentre outras aplicações. O modelo aplica técnicas de *Depthwise Separable Convolutions (DSC)* com o intuito de aprimorar o problema da destruição de informações nas camadas não lineares em blocos de convolução ao utilizar gargalos. Também foi introduzida uma nova estrutura chamada de *Inverted Residuals* (Resíduos Invertidos), presente entre as finas camadas de gargalo, para prever os dados que chegam (25, 26).

Também estão presentes as camadas de expansão intermediária, que empregam convoluções mais leves em profundidade para filtrar fontes de não linearidade. É crucial remover as não linearidades nas camadas estreitas para melhorar o desempenho do modelo (25).

3.2.1 *Depthwise Separable Convolution (DSC)*

As DSCs são blocos fundamentais para a construção de diversas arquiteturas de redes neurais eficientes. A ideia principal é empregar uma versão fatorizada que permita dividir a convolução em duas camadas distintas, em vez de um operador convolucional completo, que exige um maior poder computacional (25).

A camada inicial é denominada convolução em profundidade, a qual executa uma filtragem suave ao empregar um filtro de convolução básico em cada canal de entrada. A

segunda camada é uma camada de convolução 1x1, conhecida como convolução pontual, que pode gerar novos recursos por meio de combinações lineares dos canais de entrada (25).

Em convoluções convencionais, um tensor de entrada L_i com dimensões $h_1 \times w_i \times d_i$ (altura x largura x dimensão), é processado por um *kernel* convolucional $K \in R^{k \times k \times d_i \times d_j}$ para gerar um tensor de saída L_j com dimensões $h_i \times w_i \times d_j$. O custo computacional desse tipo de procedimento é $h_i \times w_i \times d_j \times k \times k$ (25).

Na convolução padrão, a camada de convolução é parametrizada por um *kernel* K de tamanho $D_k \times D_k \times M \times N$. D_k representa a dimensão espacial do *kernel*, e M indica o número de canais de entrada e N representa o número de canais de saída. Além disso, uma convolução padrão recebe um mapa de características F de $D_F \times M$ para produzir um mapa de características G de $D_F \times D_F \times N$ onde D_F representa a largura e a altura espacial do mapa de características de entrada (27).

O mapa de características de saída em uma convolução padrão, considerando apenas um passo no preenchimento, pode ser calculado a partir da Equação 3.8 (27).

$$G_{k,l,n} = \sum_{i,j,m} K_{i,j,m,n} \cdot F_{k+i-1,l+j-1,m} \quad (3.8)$$

Na Equação 3.8, $G_{k,l,n}$ indica o resultado da convolução na posição (k,l) do n -ésimo filtro, e $K_{i,j,m,n}$ trata-se do *kernel* de altura i , largura j e profundidade m que contém n filtros aplicados, e $F_{k+i-1,l+j-1,m}$ representa o mapa de características de entrada. Os valores resultantes da multiplicação são somados em todas as dimensões do filtro e em todos os canais de entrada da rede. Para saber o custo computacional, basta fazer o seguinte produto: $D_K \times D_K \times M \times N \times D_F \times D_F$ (27).

Ja na DSC, o mapa de características de saída, considerando apenas um filtro por canal de entrada, pode ser calculado a partir da Equação 3.9 (27).

$$\hat{G}_{k,l,m} = \sum_{i,j} \hat{K}_{i,j,m} \cdot F_{k+i-1,l+j-1,m} \quad (3.9)$$

Na Equação 3.9, $\hat{K}_{i,j,m}$ representa o *kernel* convolucional de profundidade de tamanho $D_K \times D_K \times M$. m indica o m -ésimo filtro do *kernel* aplicado ao m -ésimo canal do mapa de características de entrada F . Isso é para produzir o m -ésimo canal do mapa de características de saída filtrado $\hat{G}$. Diferente da Equação 3.8, o somatório ocorre apenas nas dimensões i e j devido ao processamento independente do canal de entrada m . Em outras palavras, o filtro $\hat{K}$ opera apenas um canal por vez, indicando que a DSC foca

apenas na parte espacial. Para calcular o custo computacional de filtragem única, basta fazer o seguinte produto: $D_K \times D_K \times M \times D_F \times D_F + M \times N \times D_F \times D_F$ (27).

Considerando a convolução como um processo de duas etapas de filtragem, é possível obter uma redução no custo computacional.

$$\begin{aligned} & \frac{D_K \cdot D_K \cdot M \cdot D_F \cdot D_F + M \cdot N \cdot D_F \cdot D_F}{D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F} \\ &= \frac{1}{N} + \frac{1}{D_K^2} \end{aligned} \quad (3.10)$$

A partir da Equação 3.10, é observável uma redução bastante significativa no custo computacional da convolução DSC em relação ao custo computacional da convolução padrão. Sabendo que o *MobileNetV2* utiliza filtros 3×3 ($D_K = 3$), o termo $\frac{1}{D_K^2}$ vira $\frac{1}{9}$ (27).

Em suma, em comparação aos modelos tradicionais, a DSC reduz significativamente o custo computacional em um quase fator de k^2 . O *MobileNetV2* emprega $k = 3$ (matriz 3x3) para diminuir o custo computacional de 8 a 9 vezes em relação às convoluções tradicionais (25).

3.2.2 *Linear Bottlenecks* (Blocos de Gargalo)

Considere uma rede neural profunda com n camadas, onde cada camada possui um tensor de entrada L_i com um tensor de ativação de dimensões $h_i \times w_i \times d_i$. Cada tensor de ativação é tratado como contêineres de $h_i \times w_i$ *pixels* de dimensão d_i . Para um conjunto de entrada de imagens reais, o conjunto de camadas de ativação, em qualquer camada L_i , constituem uma *manifold of interest* (variedade de interesse). Em outras palavras, ao analisar todos os *pixels* individuais do canal d de uma camada convolucional profunda, é perceptível que a informação representada nesses valores está, de fato, em alguma variedade que pode ser integrada a um subespaço de dimensão reduzida (25).

À primeira vista, pode-se pensar que esse fato poderia ser aproveitado diminuindo a dimensão de uma camada, o que resultaria na redução da dimensionalidade do espaço operacional. O *MobileNetV1* explorou com sucesso esse aspecto, buscando equilibrar o consumo computacional e a precisão por meio de um parâmetro multiplicador que ajusta a largura, o qual foi integrado em diversos projetos de rede neural. A técnica do multiplicador de largura possibilita a diminuição do espaço de ativação até que a variedade de interesse preencha todo esse espaço. No entanto, essa abordagem falha quando consideramos que as redes neurais convolucionais profundas têm transformações não lineares

por coordenada, como a *ReLU* por exemplo. Geralmente, a função *ReLU* é aplicada a uma linha no espaço unidimensional que produz um raio R^n no espaço, o que resulta uma curva linear por partes de n nós (25).

Em uma camada com a função *ReLU* (B_x), se o resultado for diferente de zero, a parte correspondente do espaço de entrada (x) é restringida a uma transformação linear (B_x). Em outras palavras, a habilidade das redes neurais profundas é restrita a um classificador linear no domínio da saída quando o volume é diferente de zero. Por outro lado, quando a função *ReLU* começa a colapsar o canal, inevitavelmente, toda a informação presente naquele canal é perdida. Todavia, se existirem muitos canais e uma estrutura no conjunto de funções de ativação que permita a preservação de informações em outros canais, essa estrutura passa a ser uma camada que pode integrar a entrada em um subespaço de menor dimensão do espaço de ativação. (26).

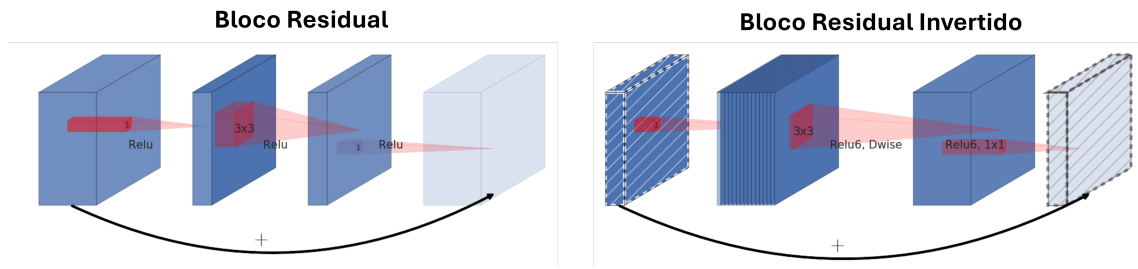
3.2.3 *Inverted Residuals* (Resíduos Invertidos)

Os blocos de gargalos assemelham-se aos blocos residuais, em que cada bloco residual contém uma entrada, seguida por vários gargalos e, posteriormente, por uma expansão. Entretanto, levando em conta que os gargalos possuem todas as informações essenciais, ao passo que a camada de expansão funciona apenas como uma implementação que acompanha uma transformação não linear do tensor, é viável empregar atalhos diretamente entre os gargalos (25).

A finalidade de se implementar esses atalhos é aprimorar a habilidade de um gradiente se propagar pelas camadas multiplicadoras. No entanto, o modelo invertido já é bastante eficaz em termos de memória (25).

A Figura 3.2 ilustra de forma esquemática a diferença entre o bloco residual e o bloco invertido. Observe que as camadas com hachuras diagonais não utilizam não-linearidades, e sim a espessura de cada bloco para indicar sua quantidade relativa de canais. Note que os resíduos clássicos ligam as camadas com uma quantidade significativa de canais, enquanto os resíduos invertidos ligam os pontos críticos (25).

Figura 3.2 – Bloco Residual vs Bloco Residual Invertido



Fonte: adaptado de (25).

3.2.4 Arquitetura

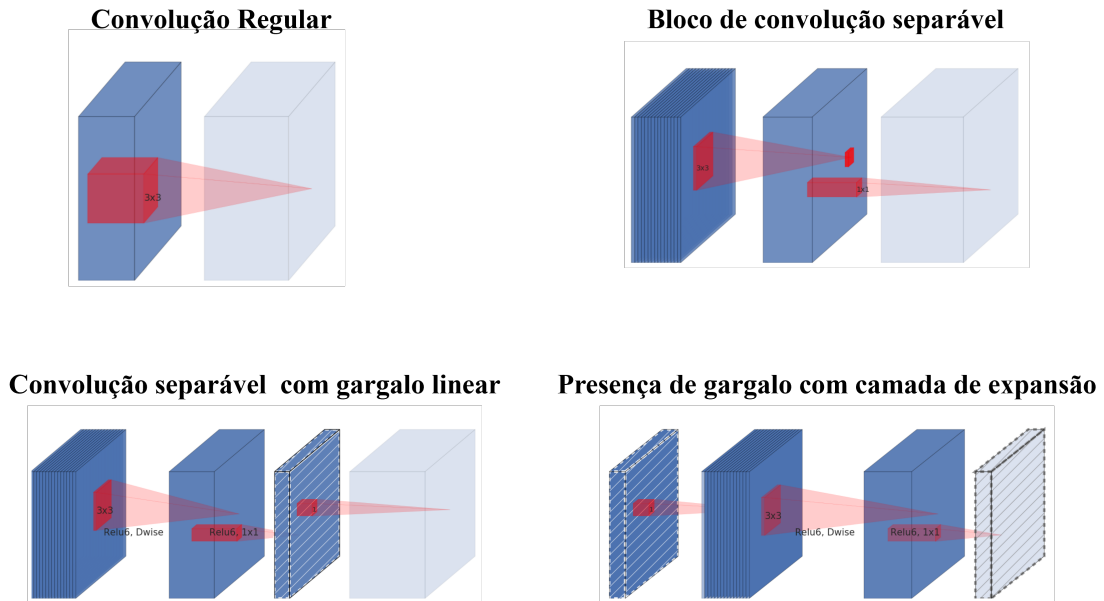
O *MobileNetV2* é formado por um bloco de convolução em profundidade com gargalo e resíduo. A estrutura detalhada do bloco DSC pode ser visualizada na Tabela 3.1 e na Figura 3.3. Considerando a transferência de aprendizado, o *MobileNetV2* pode ser visto como o principal. Podem ser usadas algumas camadas subsequentes para classificações específicas (26).

Tabela 3.1 – Bloco residual de gargalo (*bottleneck*) transformando de k para k' canais, com salto (*stride*) s e fator de expansão t .

Entrada (<i>Input</i>)	Operador	Saída (<i>Output</i>)
$h \times w \times k$	1x1 conv2d, ReLU6	$h \times w \times (tk)$
$h \times w \times tk$	3x3 dwise s=s, ReLU6	$\frac{h}{s} \times \frac{w}{s} \times (tk)$
$\frac{h}{s} \times \frac{w}{s} \times tk$	linear 1x1 conv2d	$\frac{h}{s} \times \frac{w}{s} \times k'$

Fonte: (25)

Figura 3.3 – Evolução dos blocos de convolução separável



Fonte: adaptado de (25, 26).

Na Figura 3.3, as camadas não lineares são representadas pelas hachuras diagonais. A camada final (cor clara) representa o começo do próximo bloco. Ademais, a convolução separável com gargalo linear e a convolução de gargalo com camada de expansão são, basicamente, equivalentes (25).

A arquitetura do *MobileNetV2* possui uma camada inicial de convolução completa de 32 filtros, seguida de 19 camadas residuais de gargalo (25). É possível visualizar a descrição dessas camadas na Tabela 3.2.

Tabela 3.2 – Descrição da arquitetura da rede *MobileNetV2*.

Entrada (<i>Input</i>)	Operador	t	c	n	s
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1x1	-	1280	1	1
$7^2 \times 1280$	avgpool 7x7	-	-	1	-
$1 \times 1 \times 1280$	conv2d 1x1	-	k	-	-

Fonte: (25)

Na Tabela 3.2, cada linha apresenta uma sequência de uma ou mais camadas

idênticas, que são replicadas n vezes. Em uma sequência, todas as camadas têm a mesma quantidade de canais de saída (c). Em cada sequência, o primeiro elemento contém um passo s , enquanto os demais elementos contêm apenas um passo. As convoluções espaciais empregam *kernels* de tamanho 3×3 . O fator de expansão t é sempre utilizado nas dimensões de entrada do modelo, conforme pode ser observado na Tabela 3.1 (25).

Além disso, o *MobileNetV2* emprega a *ReLU6* como função de ativação não-linear devido à sua resistência quando utilizada na computação de baixa precisão. Em redes mais avançadas, emprega-se um *kernel* 3×3 , juntamente com *dropout* e normalização em lote durante o treinamento. Em todas as camadas, exceto a primeira, uma taxa de expansão constante é aplicada em toda a rede neural. Taxas de expansão entre 5 e 10 produzem curvas de desempenho muito semelhantes. Em redes menores, o desempenho é superior com taxas de expansão um pouco menores, ao passo que em redes maiores o desempenho é ligeiramente melhor com taxas de expansão maiores (25).

Nos experimentos realizados em (25), o tamanho do tensor de entrada da rede foi multiplicado por um fator de expansão igual 6. Exemplificando, considere uma camada de gargalo que receba um tensor de entrada de 64 canais e que produza um tensor de saída com 128 canais. Nesse caso, a camada de expansão terá $64 \times 6 = 384$ canais (25).

3.3 Quick Response Super Resolution (QRSR)

No *WeChat*, foi desenvolvido um modelo de CNN de super-resolução com o objetivo de aprimorar a performance de *QR codes* de menor tamanho. Essa CNN é chamada de *Quick Response Super Resolution (QRSR)*. O modelo utiliza métodos de convolução em profundidade, como a DSC descrita na subseção 3.2.1, concatenação *DenseNet* e deconvolução (19).

3.3.1 Concatenação *DenseNet*

Imagine uma única imagem x_0 sendo processada por uma rede convolucional. Essa rede tem L camadas, em que cada uma aplica uma transformação não-linear do tipo $H_l(\cdot)$, com l representando os índices das camadas. O $H_l(\cdot)$ pode ser uma função composta de operações do tipo *Batch Normalize (BN)*, *ReLU*, *Pooling* ou convolução, gerando uma saída x_l da l -ésima camada da rede (28).

Em redes convolucionais do tipo *feed-forward*, a saída da l -ésima camada é conectada na entrada da camada seguinte ($l + 1$), dando origem a uma transição de camada

$x_l = H_l(x_{l-1})$. Com o intuito de aprimorar o fluxo de informação entre as camadas, o *DenseNet* introduz conexões diretas de qualquer camada para todas as camadas subsequentes. Consequentemente, a l -ésima camada recebe os mapas de características de todas as camadas anteriores. Isso pode ser observado na Equação 3.11 (28).

$$x_l = H_l([x_0, x_1, \dots, x_{l-1}]) \quad (3.11)$$

Na Equação 3.11, x_l indica a saída atual da camada l e $[x_0, x_1, \dots, x_{l-1}]$ representa a concatenação dos mapas de características produzidas em todas as camadas, aumentando a variação na entrada das camadas subsequentes para melhorar a eficiência. Para facilitar a implementação, é possível concatenar as múltiplas entradas de $H_l(\cdot)$ em um único tensor (28).

3.3.1.1 Taxa de crescimento

Como foi visto anteriormente, sabe-se que cada função H_l produz k mapas de características, onde a l -ésima camada possui $k_0 + k \times (l - 1)$ mapas de características na entrada. k_0 é o número de canais na camada de entrada. Assim, o hiper parâmetro k é introduzido como sendo a taxa de crescimento da rede, taxa essa que define a quantidade de mapas de características que cada camada contribui. Como as camadas são estreitas, a taxa de crescimento geral é constante, controlando a expansão para manter a eficiência da rede (28, 29).

3.3.1.2 Reutilização de recursos

Em suma, o modelo *DenseNet* permite que as camadas acessem os mapas de características das camadas precedentes onde é computado o peso médio absoluto atribuído em cada camada convolucional l . O peso médio serve para indicar a dependência de uma camada convolucional em relação às suas camadas anteriores. Além disso, todas as camadas compartilham seus pesos entre diferentes entradas dentro de um mesmo bloco, indicando que as primeiras características obtidas nas primeiras camadas são usadas nas camadas profundas no mesmo bloco (28).

Nas camadas de transição, os pesos de tais camadas são distribuídos dentro do bloco denso precedente, o que indica a presença de um fluxo de informações da primeira a última camada por meio de poucas indireções (28).

3.3.2 Deconvolução

Interpolações bilineares simples são empregadas no processo de deconvolução para conectar saídas que geram *pixels* densos. Além da interpolação, são empregadas convoluções com passo de entrada fracionário de $f = 1$ onde f é um fator de sobreamostragem. Teoricamente, realizar a sobreamostragem por um fator f é equivalente a aplicar uma convolução com passo de entrada $\frac{1}{f}$. Seja f um número inteiro, a operação torna-se uma convolução reversa com passo de saída f . Essa abordagem é bastante trivial, pois permite que a sobreamostragem seja integrada à arquitetura para aprendizado via retropropagação a partir da perda de *pixel* a *pixel* (30).

Diferente de outros métodos, os filtros de deconvolução não são predefinidos, e sim otimizados durante o treinamento, permitindo que a rede aprenda a função de sobreamostragem não linear mais adequada para restaurar a nitidez dos módulos dos códigos (30).

Em outras palavras, queremos recuperar de uma imagem interpolada de baixa resolução $\mathbf{Y}$ uma imagem $F(\mathbf{Y})$ que seja a mais próxima possível de uma imagem real $\mathbf{X}$ de alta resolução. Para isso, é necessário construir um mapeamento F , que teoricamente consiste em três passos: extração e recuperação de *patches*, mapeamento não linear e reconstrução (31).

3.3.2.1 Extração e recuperação de *patches*

A rede extrai pequenas regiões sobrepostas da imagem de baixa resolução $\mathbf{Y}$, as quais são representadas como um vetor de alta resolução. Tais vetores consistem em um conjunto de mapas de características, sendo a quantidade deles igual à dimensão dos vetores. Aqui o foco é identificar as bordas e cantos do *QR code* (31).

3.3.2.2 Mapeamento não linear

Nessa operação, cada um dos vetores de alta resolução é mapeado não linearmente em outro vetor de alta resolução. Cada um desses vetores é a representação de um *patch* de alta resolução. Isso é feito para garantir que nenhuma informação seja perdida durante o mapeamento (31).

3.3.2.3 Reconstrução

Nesse processo, é necessário agregar todas as representações de alta resolução do vetor criado pelo mapeamento não linear, *patch a patch*, para gerar a imagem final de alta resolução $F(\mathbf{Y})$. Esta imagem deve se assemelhar à imagem $\mathbf{X}$ (31).

3.4 Considerações finais

Neste capítulo, foi vista a arquitetura e os algoritmos subjacentes que compõem o *WeChat QR Code Detector*, uma ferramenta de alto desempenho disponibilizada pela biblioteca *OpenCV*. A base para a detecção eficiente dos códigos nesse módulo é graças à utilização combinada da estrutura SSD acoplada a uma rede *backbone* fundamentada no *MobileNetV2*. O método SSD garante a identificação precisa das caixas delimitadoras em múltiplas escalas de forma ágil, o que supera o obstáculo de processamento em tempo real. Já a integração do *MobileNetV2* garante uma redução significativa no custo computacional, sendo indispensável para a viabilidade em dispositivos móveis, almejada por meio de inovações como as convoluções separáveis de profundidade e os blocos de resíduos invertidos.

Além das etapas de detecção primária e extração de características, foram explorados também as particularidades do modelo QRSR. Essa rede neural convolucional complementar foi desenvolvida especificamente para mitigar as limitações de códigos pequenos, danificados ou capturados em baixa resolução. A implementação da concatenação *DenseNet* para melhorar o aprimoramento de fluxo, juntamente com a deconvolução, otimizada para restaurar a nitidez dos módulos, garante que o sistema lide com cenários adversos durante a captura da imagem.

4 Material e Métodos

4.1 Método

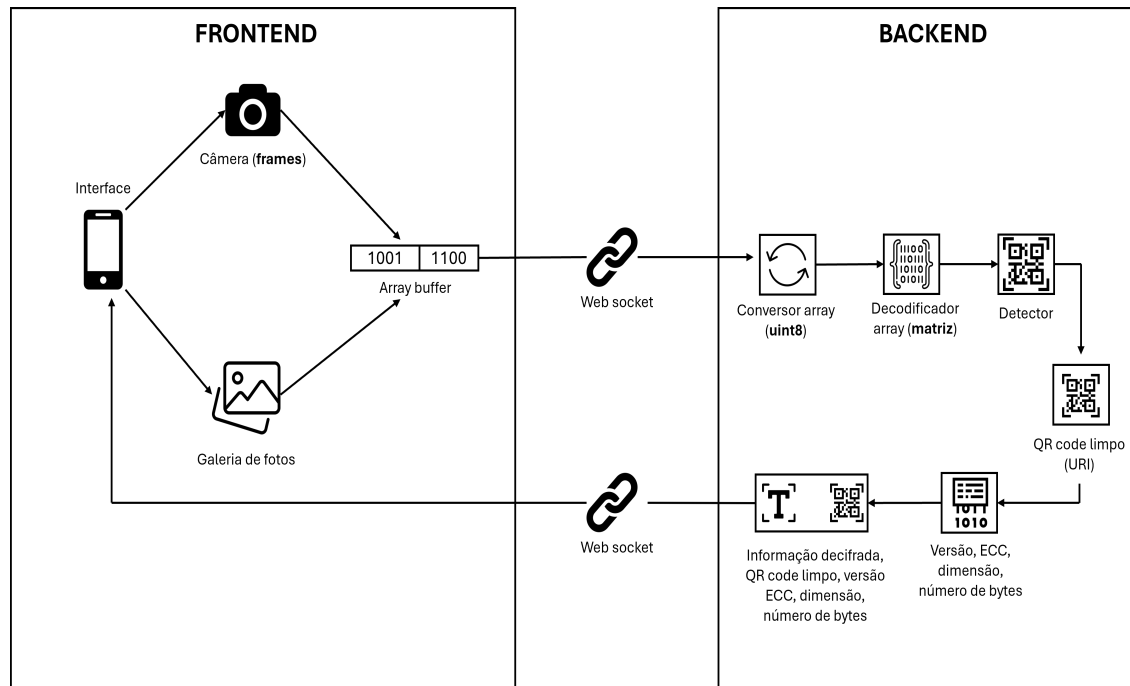
4.1.1 Aplicativo

O estudo necessário para o desenvolvimento do aplicativo para detecção e interpretação do *QR code* abrange diversas áreas da computação, com destaque para detecção de objetos. Durante a construção do projeto, foram escolhidas duas linguagens de programação: uma para desenvolver o *frontend* e outra para o *backend*, bem como o sistema operacional que vai rodar o aplicativo.

O *frontend* é a parte visível e visual do aplicativo, com o qual o usuário deve interagir com botões que permitem a obtenção de imagens que contenham códigos *QR code* através da galeria de fotos ou da câmera do celular, assim como a criação de novos códigos ao inserir textos. As imagens selecionadas, ou os frames registrados em 15 *frames* por segundo (fps) pela câmera, são transformados em um *array* de dados binários e transmitidos para um servidor por meio de uma conexão *websocket*. Se a imagem ou o *frame* conter um *QR code*, o servidor envia a informação decifrada, juntamente com outros dados como versão, ECC, dimensão, número de *bytes* e uma imagem que contém apenas o código reconstruído, permitindo o usuário visualizar os detalhes do código decifrado e salvar a imagem em sua galeria de fotos.

O servidor (*backend*) hospeda o *script* responsável pela decodificação, e utiliza dois modelos de CNN: um destinado à detecção de objetos e outro à super-resolução. Ao receber os dados binários por meio do *websocket*, é necessário convertê-los em um *array* de números inteiros de 8-bits, ou, em termos técnicos, em um *uint8 array*. Depois disso, deve-se encaminhá-lo ao decodificador, que lerá esse *array* na memória e o transformará em uma matriz (32). A matriz é encaminhada ao detector, que reconhece o código *QR code* existente, e a informação decifrada. Em seguida, é criada uma imagem limpa do *QR Code* em URI, que é transmitida ao *frontend* via *websocket*, além de outras informações detalhadas do código como versão, dimensão, número de *bytes* e ECC. É possível visualizar o diagrama completo da decodificação do sistema na Figura 4.1.

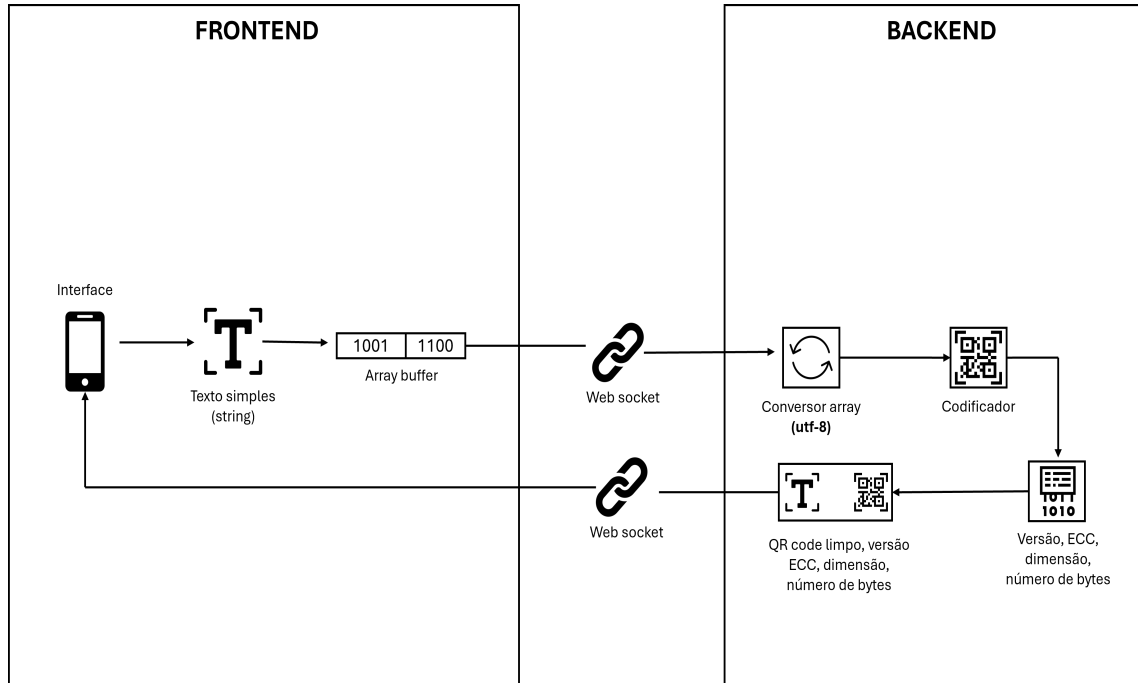
Figura 4.1 – Diagrama decodificação do sistema



Fonte: autor

Na codificação da mensagem escrita, conforme mostrado na Figura 4.2, o *frontend* converte o texto simples (*string*) em um *array* de dados binários, que é enviado ao servidor (*backend*) por meio da conexão *websocket*. No servidor, o *array* é transformado novamente em texto simples (utf-8) e enviado a uma função que codifica os dados e gera uma imagem limpa do *QR Code* em URI. O servidor envia o código limpo ao usuário através da conexão *websocket*.

Figura 4.2 – Diagrama codificação do sistema



Fonte: autor

4.1.2 Experimento

4.1.2.1 Decodificação e detecção

Com o objetivo de identificar as limitações do detector, foram realizados diversos testes. Inicialmente, variou-se a altura do suporte onde o celular foi posicionado para realizar a leitura dos códigos. As alturas de 32 cm, 66 cm e 105 cm foram selecionadas porque, nessas posições, o suporte do celular se fixava melhor nos ajustes do tripé. Além disso, tais alturas contemplam um intervalo de distâncias comumente utilizadas para detectar *QR codes*. Outro aspecto importante, é que a iluminação foi ajustada para cada um dos *QR codes* utilizados nos experimentos em cada altura. O cenário de iluminação foi dividido em três tipos: iluminação mínima, em que a luz do laboratório e do suporte de iluminação foram apagados; iluminação parcial, em que apenas a luz do laboratório estava ligada; e iluminação máxima, em que ambas as luzes estavam ligadas.

Para obter as versões de *QR code* desejadas, cada um dos *QR codes* foi criado por meio de um *script* que gera o código, repetindo o caractere "a" até alcançar a versão requerida. Para o experimento, selecionamos quatro variantes do *QR code*: versão 1, versão 10, versão 20 e versão 40. Em cada versão, foram produzidos quatro códigos com diferentes níveis de correção de erro (L, M, Q e H), resultando em um total de 16 imagens. Considerando os três cenários de iluminação, foram feitos 48 testes nessa análise. Além desses

códigos, também foram gerados mais quatro códigos de cada nível de correção de erro, que incluem o *link* para o Instagram da página oficial do curso de Engenharia Eletrônica e de Telecomunicações da Universidade Federal de Uberlândia, campus Patos de Minas.

Além da altura e da iluminação, foi analisado também o impacto da rotação na detecção e decodificação nos códigos. Para essa análise, mantivemos a altura e a iluminação constantes, pois o objetivo é avaliar apenas a influência da rotação no cenário com as maiores taxas de sucesso na detecção dos códigos, e os *QR codes* foram rotacionados nos ângulos de 10°, 20°, 30° e 40°.

4.1.2.2 Codificação

Com o objetivo de demonstrar a codificação do aplicativo, foram realizados alguns testes com textos de tamanhos variados. No experimento, conforme mostrado na Tabela 4.1, utilizamos o *link* da referência (19), o título do trabalho, o resumo escrito no Projeto Final de Curso 1 e um parágrafo do Capítulo 3.

Tabela 4.1 – Dados utilizados nos testes de codificação do aplicativo

Informação	Tipo	Tamanho (Bytes)
https://github.com/opencv/opencv_contrib/tree/4.x/modules/wechat_qrcode	Link	71
Uma análise detalhada do funcionamento de <i>QR codes</i>	Texto	51
Entre os 2500 algoritmos, há um detector de <i>QR code</i> de alto desempenho amplamente empregado em vários aplicativos da Tencent, como as redes sociais chinesas <i>WeChat</i> e <i>WeCom</i> .	Texto	~174
Este trabalho descreve o desenvolvimento de um aplicativo para facilitar a obtenção de dados através da leitura de códigos <i>QR code</i> . O aplicativo irá escanear o <i>QR code</i> com a câmera do celular e fornecer a informação contida, a versão e o nível de correção de erro do código. Para a construção do servidor que vai hospedar o modelo de decodificação do aplicativo, serão utilizadas as bibliotecas <i>OpenCV</i> , <i>Numpy</i> e <i>WebSockets</i> da linguagem <i>Python</i> , enquanto o <i>framework React Native</i> da linguagem <i>Javascript</i> será utilizado no desenvolvimento <i>frontend</i> . Serão realizados testes para verificar a precisão do aplicativo. Além do aplicativo, o projeto busca disseminar o conhecimento a respeito do <i>QR code</i> .	Texto	~716

Fonte: autor

A quantidade de *bytes* dos textos mostrados na Tabela 4.1 foi determinada por meio de um *script* em *Python*, empregando o código *utf-8*.

4.2 Material

4.2.1 Aplicativo

O aplicativo foi desenvolvido para funcionar em dispositivos *android*. O *android* oferece algumas vantagens como processo de aprovação mais rápido, flexibilidade na publicação, custos de desenvolvimento mais baixos e integração com vários ecossistemas.

No desenvolvimento do aplicativo, tanto no *frontend* quanto no *backend*, o *Visual Studio Code*, editor de código leve, multiplataforma, gratuito e de código aberto mantido pela Microsoft (33), foi fundamental para a criação dos *scripts* em *Javascript* e *Python* e na renderização do aplicativo.

A parte visual do aplicativo, o *frontend*, foi construído utilizando uma estrutura de aplicativo móvel baseada na linguagem de programação *Javascript*. Essa estrutura, conhecida como *React Native*, permite a construção de aplicativos móveis que são renderizados nativamente para *android* e *IOS* (34).

Durante a fase de testes, a renderização do aplicativo foi possível graças ao *framework Expo*. O *Expo* é um *framework* que facilita o desenvolvimento de aplicativos *android*, oferecendo o roteamento baseado em arquivos, contendo uma biblioteca padrão de módulos ativos. Para utilizar tais recursos do *Expo*, foi necessário instalar o aplicativo *Expo Go* para renderizar o código do aplicativo leitor de *QR code* (35).

O desenvolvimento do servidor, encarregado de decodificar a informação contida no *QR code* e codificar a informação fornecida pelo usuário, foi realizado utilizando a linguagem de programação *Python*. Trata-se de uma linguagem de programação interpretada, orientada a objetos e de alto nível, que combina tipagem e vinculação dinâmica (36).

Ao receber a informação em *bytes*, o *backend* converte esses *bytes* em um *uint8* para processar a imagem, utilizando a biblioteca *Numpy*. Em seguida, utilizando o *OpenCV*, é necessário converter o *array numpy* em uma matriz do tipo *MatLike*. A matriz é encaminhada ao decodificador do *WeChat* para verificar se o *frame* da imagem recebida contém algum *QR code* presente na imagem. Em caso afirmativo, a função do decodificador retorna uma variável do tipo *string*, representando a informação contida no código. Essa informação é utilizada para construir um *QR code* limpo e obter a versão do código presente no *frame*, utilizando a biblioteca *qrcode*. Além da versão e do código limpo, é utilizada a biblioteca *xingcpp* para obter a versão do nível de correção de erro do código. Foi necessário utilizar essas duas bibliotecas pois o *WeChat* só fornece a informação contida no *QR Code*. Após decodificar a informação, construir o código limpo, obter a versão e o nível de correção de erro do *QR code*, esses dados precisam ser transmitidos ao *frontend* utilizando uma conexão *websocket* através da biblioteca *WebSocket*.

Além da decodificação, o servidor também codifica o texto fornecido pelo usuário em um *QR code*. A biblioteca *qrcode* é responsável pela codificação e geração de um *QR code* limpo, que é enviado ao usuário através da conexão *websocket*.

4.2.2 Experimento

Na realização do experimento, usou-se um tripé com cerca de 1,10m de altura e um suporte para celular. O tripé têm três ajustes de altura: 32cm, 66cm e 105, sendo

essas alturas utilizadas nos três cenários. O suporte é montado no tripé de modo que o celular permaneça estável à medida que variamos a posição do mesmo.

Além desses materiais, usamos também uma luminária *softbox* pequena para diversificar a iluminação do local do experimento. A Figura 4.3 ilustra a montagem feita para realizar o experimento.

Figura 4.3 – Montagem do experimento

a) Montagem completa



b) Zoom da montagem



Fonte: autor

Para executar o aplicativo, conforme apresentado na Figura 4.3, utilizou-se um celular Samsung modelo S20, que estava ligado ao computador via cabo USB, permitindo que o *Expo Go* realizasse a renderização do aplicativo. É possível verificar as especificações técnicas da câmera do celular na Tabela 4.2.

Tabela 4.2 – Especificações técnicas da câmera do celular Samsung Galaxy S20

Especificações	Conteúdo
Megapixel	12 Mp + 64 Mp + 12 Mp
Resolução	4000 x 3000 pixel
Aperture Size	F 1.8 + F 2 + F 2.2
Estabilização	Ótica
Ângulo Máximo	120°
Zoom Ótico	3x
Flash	LED
Câmera frontal	10 Mp F 2.2

Fonte: (37)

4.3 Considerações Finais

O desenvolvimento do aplicativo para detecção e interpretação de códigos *QR code* abrange várias áreas da computação, com destaque para detecção de objetos. No decorrer do projeto, optou-se por duas linguagens de programação: *Javascript* e *Python* para o desenvolvimento do *frontend* e *backend*. Além das linguagens de programação, optou-se construir o aplicativo para operar em dispositivos *android*.

O *frontend*, parte visível e visual do aplicativo, foi construído utilizando uma estrutura de aplicativo móvel conhecida como *React Native*. O *React Native* permite a construção de aplicativos que são renderizados nativamente para *android*. Para renderizar o aplicativo na fase de testes, o *framework Expo* foi crucial para o correto funcionamento do aplicativo. A utilização dos recursos do *Expo* foi possível ao instalar o aplicativo *Expo Go*.

No *backend*, as bibliotecas *Numpy*, *OpenCV*, *WebSocket*, *qrcode* e *xingcpp* são utilizadas na detecção e decodificação, além da codificação, do *QR code*.

Na seção experimental, diversos testes foram conduzidos para identificar as limitações do codificador e detector. No decodificador, esses testes envolvem a variação da altura do celular e da iluminação do ambiente, bem como a rotação dos *QR codes*. Já no codificador, os testes foram realizados utilizando *links* e textos de diferentes tamanhos na construção dos códigos requeridos.

5 Resultados e Discussões

Neste capítulo são apresentados os resultados obtidos da decodificação e detecção para cada uma das quatro versões do *QR code* escolhidas em cada um dos quatro níveis de correção de erro (L, M, Q e H), variando o nível de iluminação nas alturas de 32cm, 66cm e 105cm. Os resultados apresentados são a informação decodificada, a versão, o número de *bytes*, a dimensão e o ECC dos códigos gerados. Esses dados são mostrados nos *prints* da tela do aplicativo em questão. Além disso, são apresentados também os resultados obtidos ao variar a rotação de todos os códigos gerados, sendo a versão 40 o foco principal.

Além da detecção e decodificação, são mostrados também os resultados obtidos da codificação do *link* e dos três textos escolhidos com tamanhos variados. Os resultados apresentados correspondem aos *QR codes* criados pelo aplicativo ao inserir os dados de texto no aplicativo.

Com os resultados em mãos, é possível avaliar as possíveis limitações do codificador e do modelo de detecção do *WeChat* ao identificar um *QR code*, desde o mais simples (versão 1) até o mais avançado (versão 40), em diversos contextos.

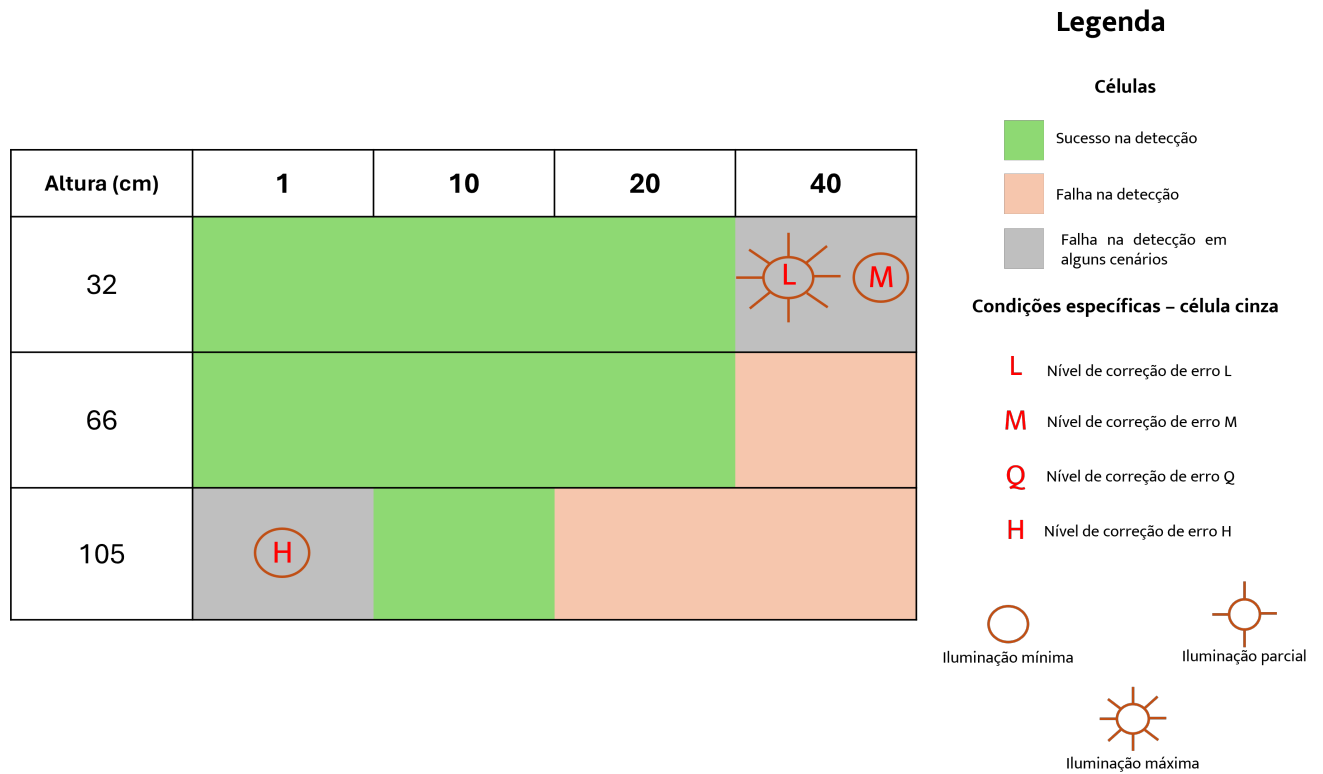
5.1 Resultados do experimento

5.1.1 Decodificação e detecção

5.1.1.1 Altura e iluminação

Na primeira análise, variamos a altura, isto é, a distância entre a câmera do celular e o *QR code*, e as condições de iluminação. Conforme mencionado na seção de metodologia, essa análise resultou em mais de 48 cenários de testes. Dessa forma, é interessante sintetizarmos os resultados encontrados nesse experimento a fim de facilitar o entendimento dos mesmos. A Figura 5.1 condensa os resultados encontrados. É possível visualizar rapidamente que versões maiores de *QR code*, bem como distâncias maiores, resultam em maior dificuldade de reconhecimento. Além disso, as células em cinza destacam situações que serão discutidas a seguir.

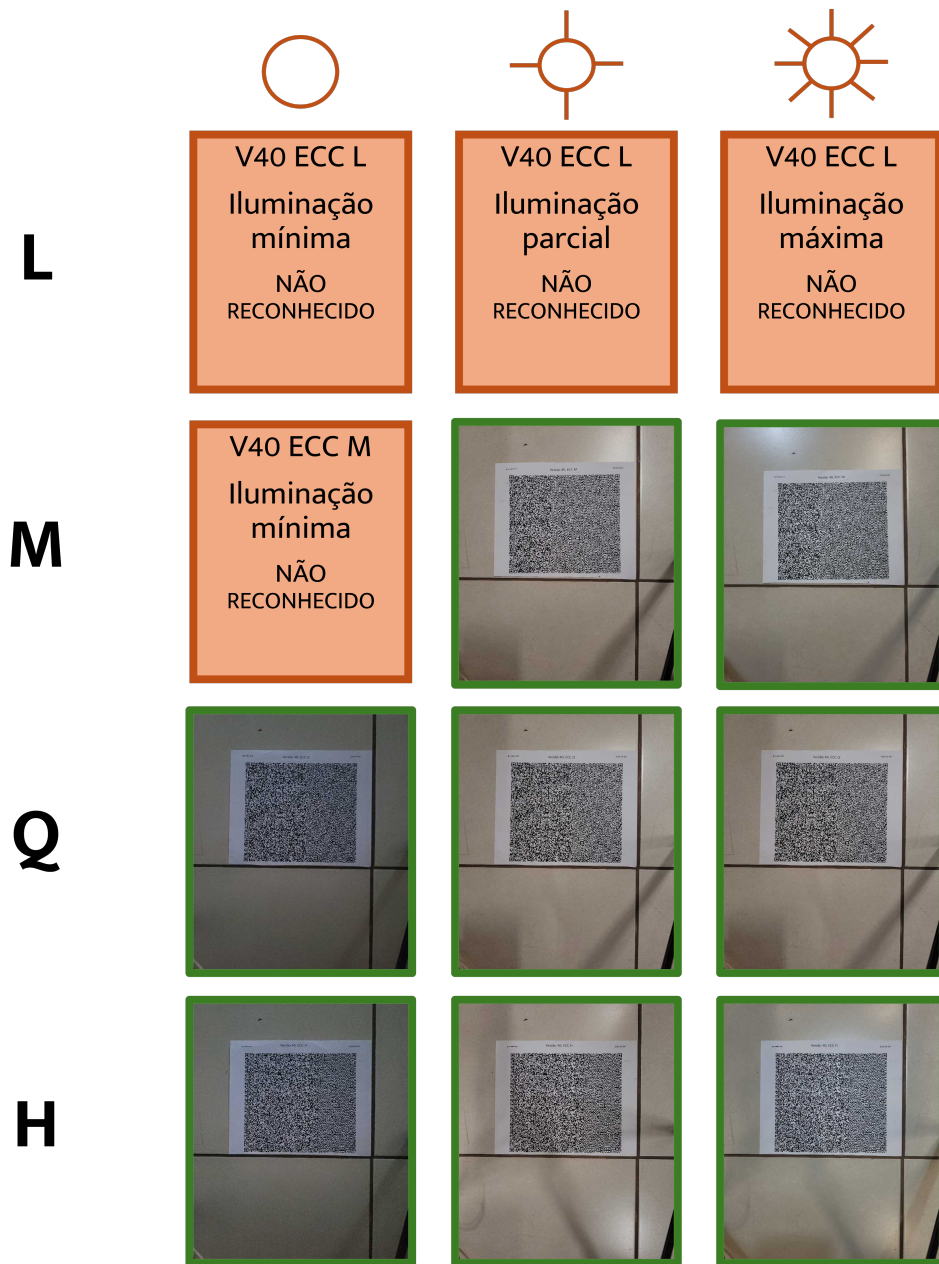
Figura 5.1 – Resultados obtidos do experimento



Fonte: autor

Na altura baixa, todas as versões foram reconhecidas pelo aplicativo, exceto alguns *QR codes* da versão 40. Para o *QR code* da versão 40 ECC L, não houve reconhecimento em nenhuma condição de iluminação, isto é, nem mesmo na condição mais iluminada. Já para o ECC M, o *QR code* só não foi reconhecido na condição de iluminação mínima. A Figura 5.2 mostra as fotos para os *QR codes* da versão 40 que foram reconhecidos e destaca em laranja em que não houve detecção.

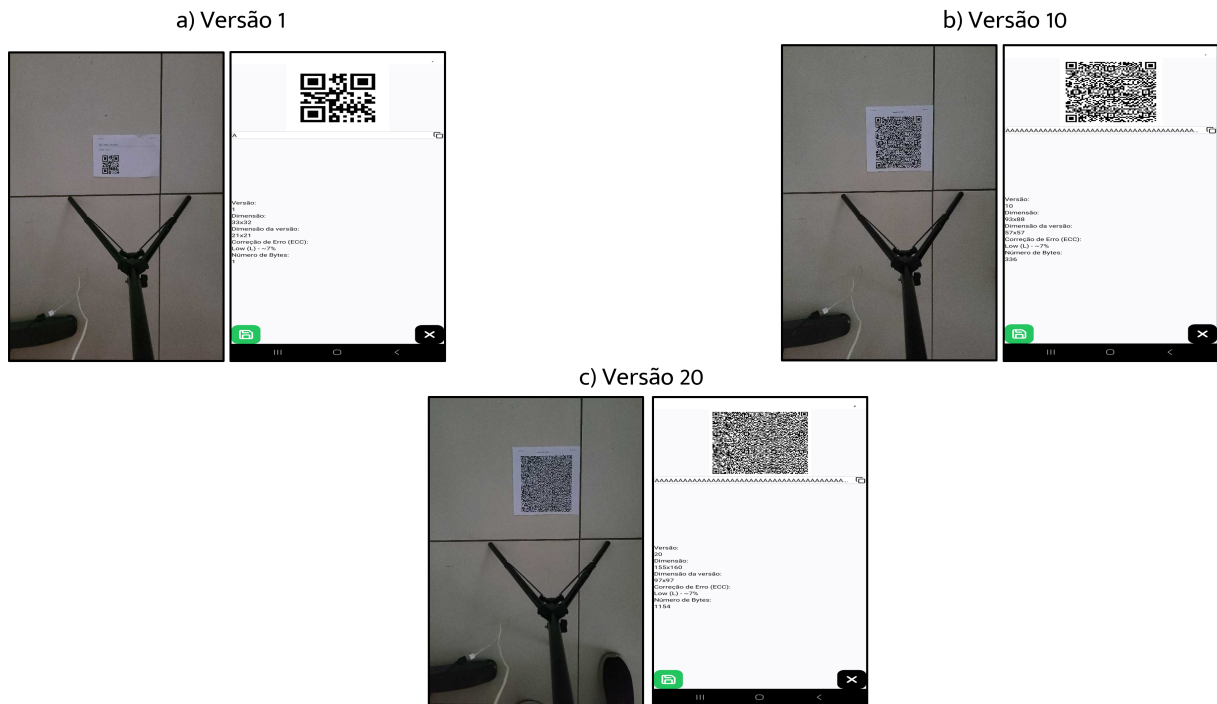
Figura 5.2 – *QR codes* da versão 40 reconhecidos pelo aplicativo na altura mínima



Fonte: autor

Para a altura média, nenhum *QR code* da versão 40 foi reconhecido em qualquer condição de iluminação. Conforme é apresentado na Figura 5.3, todas as outras versões foram reconhecidas, mesmo nas situações mais adversas, isto é, de baixa iluminação e ECC L.

Figura 5.3 – *QR codes* das versões 1, 10 e 20 detectados nas condições mais adversas



Fonte: autor

Para a altura máxima, além da versão 40, a versão 20 também não teve *QR codes* reconhecidos. Além disso, para a versão 1, com o nível de correção de erro H, o aplicativo não foi capaz de reconhecer o *QR code* na condição de baixa iluminação. Na Figura 5.4 destacam-se os *QR codes* da versão 1 reconhecidos.

Figura 5.4 – *QR codes* da versão 1 reconhecidos pelo aplicativo na altura máxima



Fonte: autor

5.1.1.2 Rotação

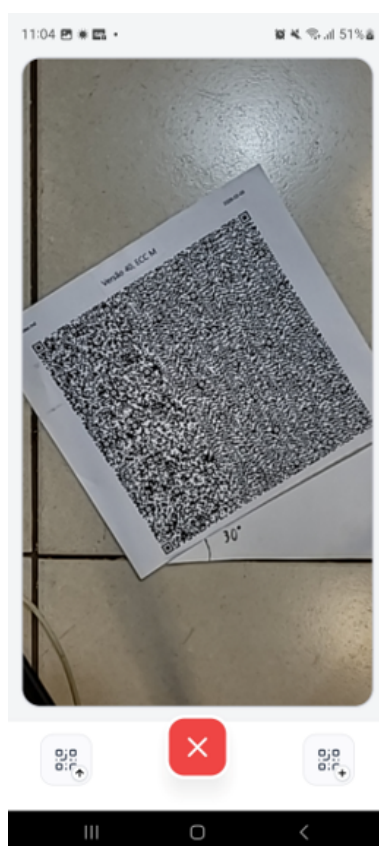
Para verificar a influência da rotação na detecção, manteve-se a iluminação no máximo e a altura em 33,5cm para melhorar o enquadramento dos *QR codes* maiores quando estavam rotacionados.

Em todos os cenários de rotação, o aplicativo foi capaz de identificar todos os *QR*

codes das versões 1, 10 e 20, abrangendo também os *QR codes* que incluem o *link*. Porém, na versão 40, a situação muda completamente.

Na versão 40, os *QR codes* com níveis de correção de erro L e M não foram detectados em todos os ângulos. A Figura 5.5 ilustra o *QR code* versão 40 ECC M não reconhecido pelo aplicativo.

Figura 5.5 – *Print* da tela do aplicativo com o *QR code* versão 40 ECC M não reconhecido em um ângulo de 10°

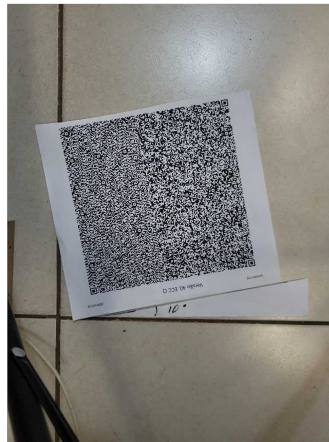


Fonte: autor

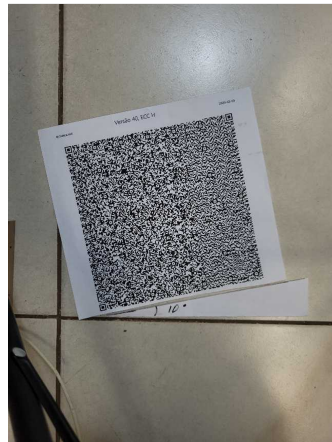
Nos níveis de correção Q e H, o aplicativo conseguiu detectar os *QR codes* em um ângulo de 10°, conforme ilustrado na Figura 5.6. No entanto, o aplicativo não foi capaz de identificá-los a partir do ângulo de 20°. A Figura 5.7 ilustra o *QR code* versão 40 ECC H não reconhecido pelo aplicativo.

Figura 5.6 – *QR codes* versão 40 reconhecidos no ângulo de 10°

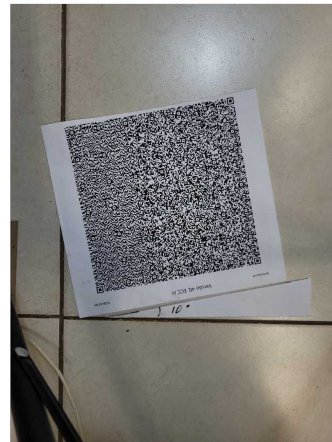
a) ECC Q (cabeça para baixo)



b) ECC H

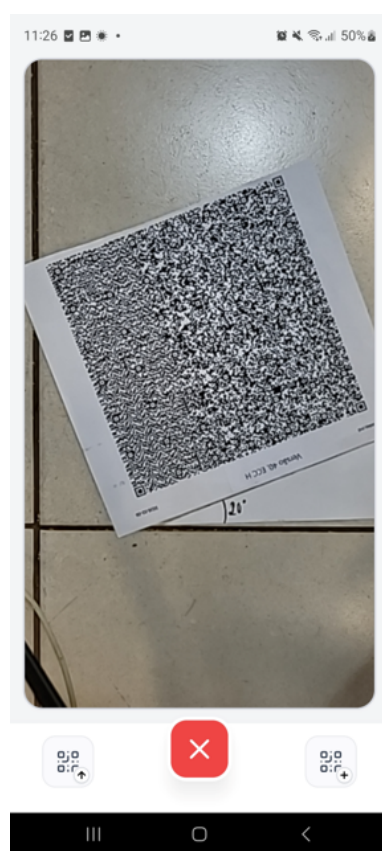


c) ECC H (cabeça para baixo)



Fonte: autor

Figura 5.7 – *Print* da tela do aplicativo com o *QR code* versão 40 ECC H não reconhecido em um ângulo de 20°

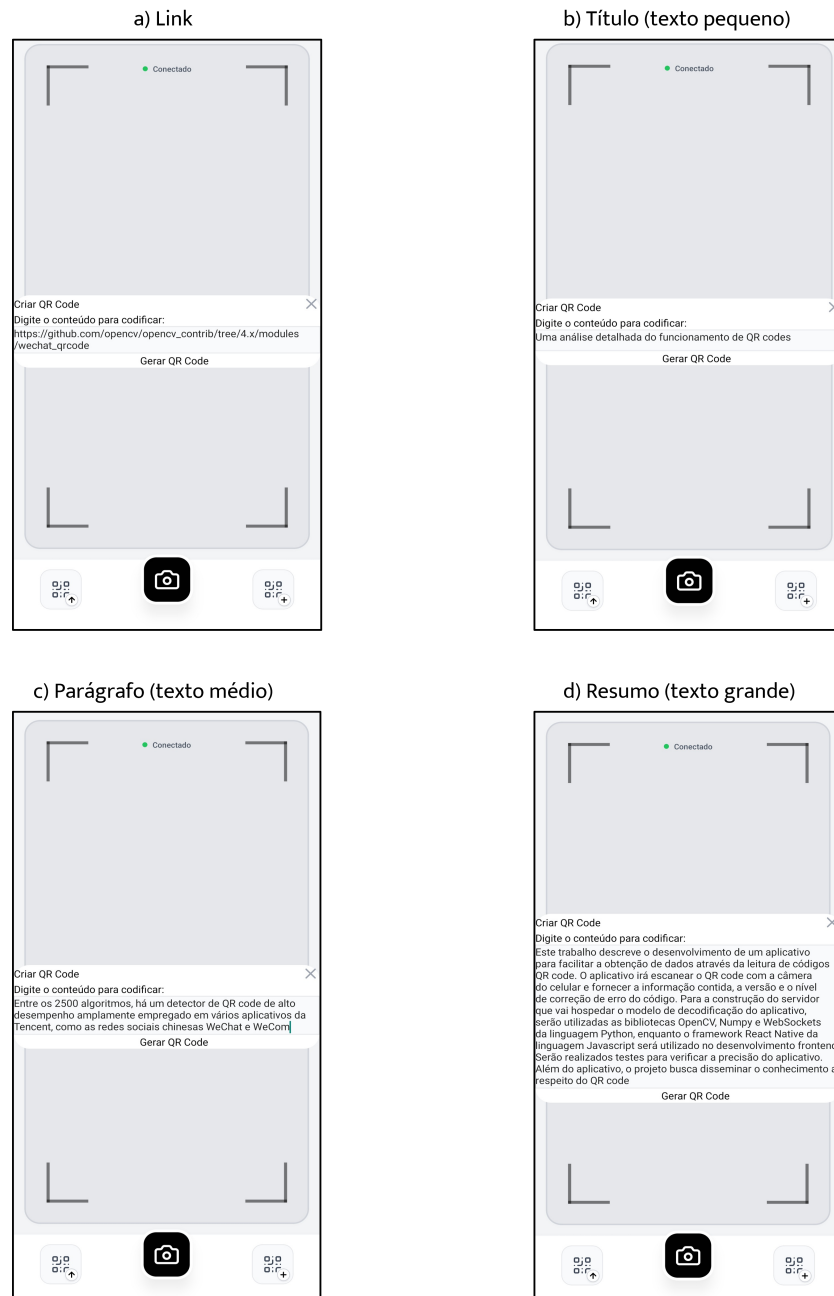


Fonte: autor

5.1.2 Codificação

Primeiramente, os dados foram inseridos na aba de textos do aplicativo conforme mostrado na Figura 5.8.

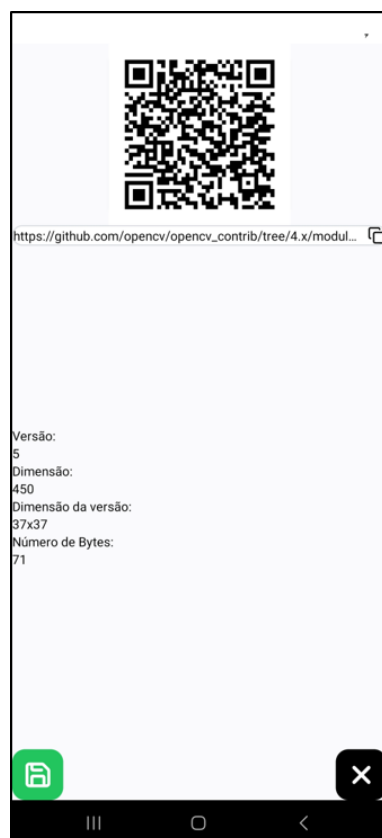
Figura 5.8 – Inserção das requeridas informações na seção de textos do aplicativo



Fonte: autor

Para o *link*, o aplicativo conseguiu gerar o *QR code* conforme mostrado na Figura 5.9.

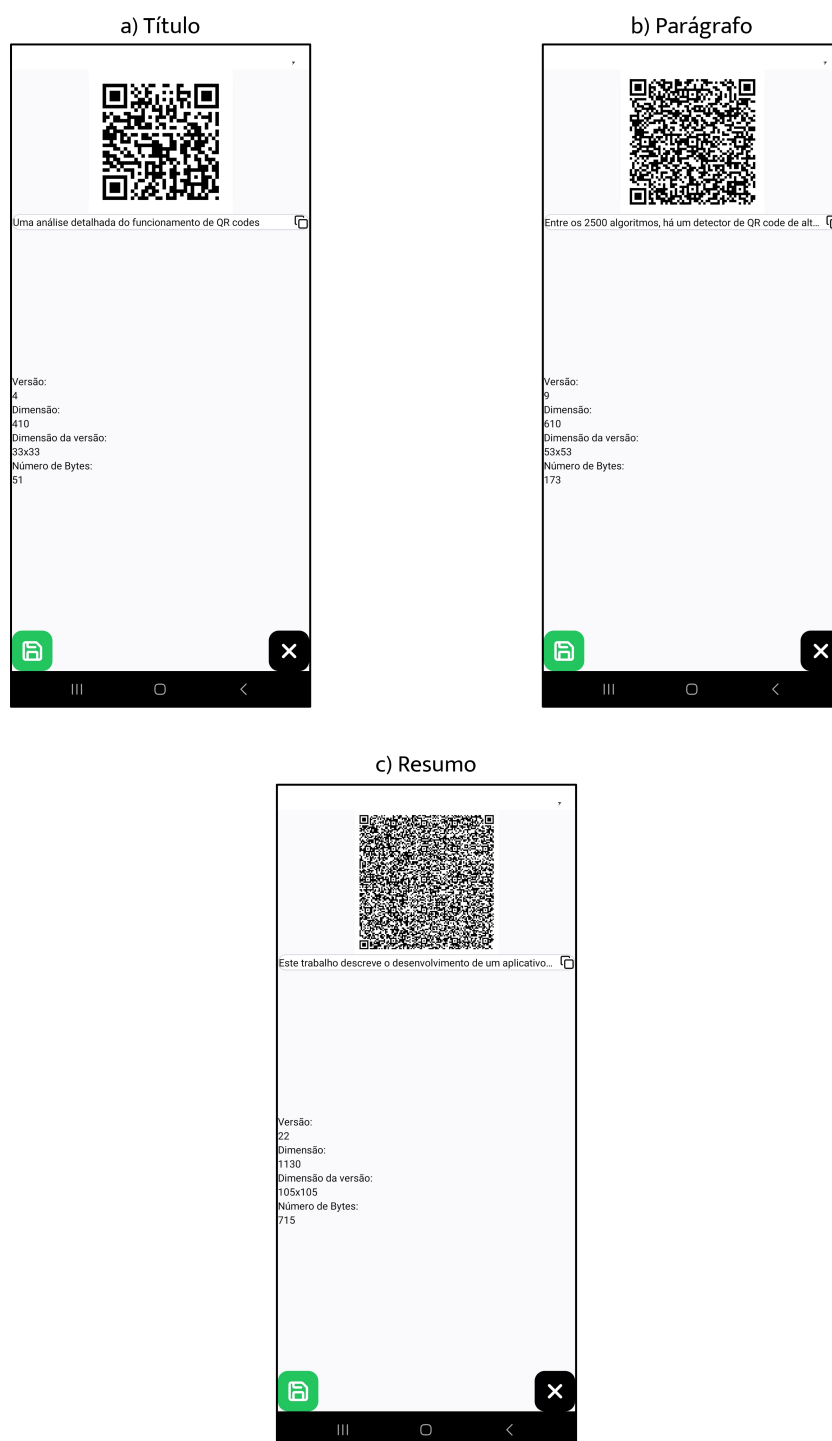
Figura 5.9 – *Link* codificado pelo aplicativo



Fonte: autor

Já nos três textos, o aplicativo também conseguiu gerar os *QR codes* dos mesmos como mostra a Figura 5.10.

Figura 5.10 – Textos codificados pelo aplicativo



Fonte: autor

5.2 Considerações finais

Neste capítulo foram apresentados os resultados obtidos da decodificação e detecção para cada uma das quatro versões do *QR code* escolhidas em cada um dos quatro níveis de correção de erro, variando o nível de iluminação nas três alturas escolhidas. Além

de analisar a influência da altura e iluminação, é mostrado também os resultados obtidos ao variar a rotação.

A análise do experimento de decodificação gerou mais de 48 cenários de teste. Assim, sintetizamos os resultados obtidos com o intuito de facilitar o entendimento dos mesmos. As células em cinza da Figura 5.1 destacam situações em que houve falhas na detecção em alguns cenários de iluminação, o quais foram discutidos em detalhes.

Além da detecção, analisamos a codificação do aplicativo, selecionando um *link* de uma das referências e três textos de tamanhos variados do artigo.

6 Conclusão

Atualmente, grande parte dos indivíduos sabem como obter ou inserir informações no *QR code* utilizando diversos aplicativos, mas desconhecem a estrutura e os processos envolvidos na codificação e decodificação desses dados. Ademais, embora muitos aplicativos consigam detectar e interpretar *QR codes* usando a câmera do *smartphone*, há alguns fatores que podem interferir na obtenção dos dados, como iluminação, distância, rotação e volume de informação. Com base nessas indagações, este projeto teve como objetivos principais disseminar o conhecimento sobre a estrutura e os processos de codificação e decodificação do *QR code*, além de construir um aplicativo móvel capaz de detectar *QR codes* por meio da câmera do celular, decodificar as informações contidas nesses códigos e codificar os dados fornecidos pelos usuários. Além de alcançar os objetivos gerais descritos anteriormente, também foi possível conduzir testes práticos no aplicativo para avaliar sua funcionalidade em diversos cenários de distância, iluminação e rotação, utilizando *QR codes* de diferentes versões e níveis de correção de erro.

Ao revisar a teoria sobre *QR codes*, redes neurais, lógica de programação e desenvolvimento de aplicativos *mobile*, foi possível criar uma ferramenta funcional que possibilita a transmissão e o processamento eficiente dos dados. Isso foi almejado ao integrar o *framework React Native* para o *frontend* em um servidor desenvolvido em *Python*, que identifica *QR codes* nas imagens e codifica as informações fornecidas pelos usuários para gerar novos códigos.

A realização dos experimentos práticos foram cruciais para revelar as potenciais limitações do modelo de detecção selecionado. Apesar dos testes serem limitados, os experimentos permitem analisar como alguns parâmetros afetam no reconhecimento de *QR codes* em situações cotidianas, em que os códigos podem estar em diferentes distâncias e condições de iluminação, bem como rotacionados.

Na altura mais baixa testada, os códigos da versão 40 com nível de correção de erro L, projetado para restaurar aproximadamente 7% dos dados corrompidos, não foram reconhecidos nem mesmo sob condições de iluminação alta. Tal análise sugere que, provavelmente, os *frames* capturados pela câmera do celular foram distorcidos, corrompendo uma quantidade de dados muito superior à capacidade mínima oferecida por esse nível de correção de erro. Já para o código da versão 40 com nível de correção de erro M, projetado para restaurar aproximadamente 15% dos dados corrompidos, não foi reconhecido

na condição de baixa iluminação. O resultado sugere que, para matrizes muito densas capturadas em distâncias menores ou em ambientes com pouca luz, a capacidade de recuperação de 7% ou 15% foi insuficiente para o aplicativo reconhecer os códigos da versão 40 ECC L (iluminação baixa, média e alta) e versão 40 ECC M (iluminação baixa).

Para altura média, a análise dos resultados obtidos sugere que houve uma limitação puramente óptica e de resolução do sistema de captura em relação à densidade estrutural do *QR code*. Como a versão 40 é uma matriz densa de 177x177 módulos, o tamanho físico do código impresso fica reduzido de forma considerável. Quando a lente foi afastada para tal distância, provavelmente, a câmera não conseguiu distinguir de forma nítida as bordas e transições finas entre os módulos claros e escuros, fazendo com que os *pixels* se misturassem e dificultando a rede neural CNN do *WeChat* de extrair as características fundamentais da imagem e interpretar os padrões de detecção e alinhamento dos códigos.

Na altura máxima, a análise dos resultados sugere que houve limitações ópticas do sistema de captura em relação à distância focal. Nesse cenário, não só a versão 40 continuou sem ser detectada, mas a versão 20 também apresentou falha total em todos os cenários de iluminação. Como a matriz geométrica dessas versões possui módulos reduzidos devido à sua complexidade, é bem provável que o sensor da câmera não foi capaz de distinguir o contraste nítido entre os blocos claros e escuros. Dessa forma, a rede neural não pode extrair as características espaciais necessárias para reconhecer a presença de um *QR code* no *frame*, o que impediu a decodificação de tais códigos. Além disso, o *QR code* na versão 1 ECC H não foi reconhecido pelo algoritmo, falhando em todos os cenários de iluminação. O resultado sugere que a complexidade da estrutura exigida pelo nível H, que concentra uma maior taxa de redundância na matriz reduzida de 21x21 módulos da versão 1, deve ter ultrapassado o limite de extração de caracteres da rede neural quando o objeto está muito distante da câmera. Como o aplicativo falhou ao detectá-lo em todos os cenários de iluminação, é bem provável que o fator crítico da falha foi a perda geométrica de definição dos módulos, e não somente o contraste luminoso.

Em relação ao impacto da rotação, os resultados obtidos mostram que o aplicativo demonstrou robustez ao ler as versões 1, 10 e 20 em todos os ângulos avaliados. Entretanto, a versão 40 com níveis de correção de erro Q e H apresentou falhas de detecção em inclinações a partir de 20°. Isso sugere que o ângulo limite desses *QR codes* varia de 10° a 20°.

Os testes práticos de codificação foram fundamentais para validar a eficiência da

ferramenta na geração de novos *QR codes*. O aplicativo processou quatro conjuntos de dados com tamanhos distintos, sendo um *link* de 71 *bytes*, o título do trabalho de 51 *bytes*, um parágrafo médio de aproximadamente 174 *bytes* e o resumo do Projeto Final de Curso 1 com cerca de 716 *bytes*. O aplicativo obteve êxito ao compilar todas as informações fornecidas, demonstrando capacidade de adaptação ao gerar matrizes estruturadas nas versões 5, 3, 8 e 22.

Conforme as análises realizadas nos experimentos de codificação e detecção, para que os resultados possam ser generalizados, é necessário que os testes sejam conduzidos com uma quantidade maior de *QR codes*.

Visando melhorar o aplicativo nas próximas versões, sugere-se a expansão das análises de decodificação sob novas condições. Recomenda-se, por exemplo, a replicação dos testes de variação de altura e iluminação com todos os *QR codes* gerados, aplicando oclusões parciais. Ou seja, tampar intencionalmente frações dos códigos impressos para verificar de forma mais rigorosas as limitações do modelo de detecção e a real eficiência do nível de correção de erro na ausência de dados visuais da matriz. Além dessas análises, outra vertente promissora envolve o aprofundamento na etapa de codificação do aplicativo. Sugere-se a realização de experimentos utilizando volumes de textos progressivamente maiores, com o intuito de identificar o limite de caracteres e *bytes* que o sistema suporta antes de apresentar falhas de processamento ou renderização. De certa forma, isso propiciará à comunidade acadêmica um mapeamento completo da capacidade operacional e dos limites da arquitetura estudada.

Referências

- 1 MIZUNO, HAJIME. **A economia japonesa após a crise do petróleo e as implicações nas suas relações com a América Latina** — scielo.br. Disponível em: <https://www.scielo.br/j/rae/a/B6sZfscR6DM838PKGtsVC6p/#:~:text=Depois%20da%20II%20Guerra%20Mundial%2C%20a%20economia%20japonesa%20registrou%20um,que%20a%20dos%20pa%C3%ADses%20desenvolvidos>. Acesso em: 30 dez. 2024.
- 2 MICROSOFT. **A brief history of QR codes**. 2024. Disponível em: <https://www.microsoft.com/en-us/microsoft-365-life-hacks/privacy-and-safety/brief-history-qr-codes>. Acesso em: 30 dez. 2024.
- 3 GOVTECH SINGAPORE. **The QR Code is everywhere, but where did it come from?** 2024. Disponível em: <https://www.tech.gov.sg/media/technews/2020-09-24-the-qr-code-is-everywhere-but-where-did-it-come-from/>. Acesso em: 30 dez. 2024.
- 4 DESCARTES, René. **Discurso do Método**. [S.l.]: Editora GHI, 1637.
- 5 TIWARI, Sumit. An introduction to qr code technology. **2016 International Conference on Information Technology (ICIT)**, 12 2016.
- 6 CHEN, Eryang; LEI, Lin. Research and realization of encoding on qr code. **2016 International Conference on Information Technology (ICIT)**, 10 2012.
- 7 SUTHEEBANJARD, Phaisarn; PREMCHAIWADI, Wichian. Qr-code generator. **2010 Eighth International Conference on ICT and Knowledge Engineering**, 11 2010.
- 8 FANG, Wen-Pinn. Offline qr code authorization based on visual cryptography. **2011 Seventh International Conference on Intelligent Information Hiding and Multimedia Signal Processing**, 11 2011.
- 9 ELHAWIL, A.; ALSELLAMI, A.; TALHA, S. A.; MOHAMED, I.; AWEIN, R. Steganography technique to hide a secret message in qr code. **THE INTERNATIONAL JOURNAL OF ENGINEERING AND INFORMATION TECHNOLOGY (IJEIT)**, v. 4, n. 2, 6 2018.
- 10 KEYENCE. **Estrutura do código QR**. 2024. Disponível em: [https://www.keyence.com.br/ss/products/auto_id/barcode_lecture/basic_2d/qr/#:~:text=Um%20c%C3%B3digo%20QR%20%C3%A9%20composto,\(c%C3%B3digo%20Reed%2DSolomon\)](https://www.keyence.com.br/ss/products/auto_id/barcode_lecture/basic_2d/qr/#:~:text=Um%20c%C3%B3digo%20QR%20%C3%A9%20composto,(c%C3%B3digo%20Reed%2DSolomon)). Acesso em: 12 jan. 2025.
- 11 GARG, Gautam. **QR Code Structure: Everything you need to know**. 2015. Disponível em: <https://scanova.io/blog/qr-code-structure/>. Acesso em: 15 jan. 2025.
- 12 FUNKE, Tobias. **QR Code Basics Explained: origin, anatomy, and use cases**. 2024. Disponível em: <https://bitly.com/blog/qr-code-basics/>. Acesso em: 17 jan. 2025.

- 13 SILVA, Alberto Renan Dias da; FANTIN, Silas. A matemática do qr code. 12 2021.
- 14 TEKALE, Haridas Atmaram. A comprehensive study on qr codes. **International Journal of Advanced Research in Science, Communication and Technology (IJARSCT)**, v. 4, n. 3, 6 2024.
- 15 QR CODE.COM. **Types of QR Code**. Disponível em: <https://www.qrcode.com/en/codes/>. Acesso em: 07 fev. 2025.
- 16 DENSO WAVE. **SQRC**. Disponível em: <https://www.denso-wave.com/en/system/qr/product/sqrc.html>. Acesso em: 09 fev. 2025.
- 17 WAN, Song; LU, Yuliang; YAN, Xuehu; DING, Wanmeng; LIU, Hanlin. High capacity embedding methods of qr code error correction. **Wireless Internet**, p. 70–79, 1 2018.
- 18 OPENCV. Disponível em: <https://opencv.org/about/>. Acesso em: 09 mar. 2025.
- 19 OPENCV. Disponível em: https://github.com/opencv/opencv_contrib/tree/4.x/modules/wechat_qrcode. Acesso em: 19 jan. 2026.
- 20 WODERSCK, Axel. **zxing-cpp**. Disponível em: <https://github.com/zxing-cpp/zxing-cpp>. Acesso em: 30 jan. 2026.
- 21 LIU, Wei; ANGUELOV, Dragomir; ERHAN, Dumitru; SZEGEDY, Christian; REED, Scott; CHENG-YANG; BERG, Alexander C. Ssd: Single shot multibox detector. **European Conference on Computer Vision**, 12 2016.
- 22 KUMAR, Vedant; GOEL, Vansh; AMORIYA, Aditya; KUMAR, Dr. Avneesh. Object detection using ssd. **International Conference on Advances in Computing, Communication Control and Networking (ICAC3N)**, 12 2023.
- 23 V7LABS. **Intersection over Union (IoU): Definition, Calculation, Code**. 2023. Disponível em: <https://www.v7labs.com/blog/intersection-over-union-guide>. Acesso em: 20 jan. 2026.
- 24 NING, Chengcheng; ZHOU, Huajun; SONG*, Yan; TANG linhui. Inception single shot multibox detector for object detection. **Proceedings of the IEEE International Conference on Multimedia and Expo Workshops (ICMEW)**, 7 2017.
- 25 SANDLER, Mark; HOWARD, Andrew; ZHU, Menglong; ZHMOGINOV, Andrey. Mobilenetv2: Inverted residuals and linear bottlenecks. **2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)**, 6 2018.
- 26 DONG, Ke; ZHOU, Chengjie; RUAN, Yihan; LI, Yuzhi. Mobilenetv2 model for image classification. **2nd International Conference on Information Technology and Computer Application (ITCA)**, 12 2020.
- 27 HOWARD, Andrew G.; ZHU, Menglong; CHEN, Bo; KALENICHENKO, Dmitry; WANG, Weijun; WEYAND, Tobias; ANDREETTO, Marco; ADAM, Hartwig. Mobilenets: Efficient convolutional neural networks for mobile vision applications. 4 2017.

- 28 HUANG, Gao; LIU, Zhuang; MAATEN, Laurens van der. Densely connected convolutional networks. 7 2017.
- 29 C., Basant. **How DenseNet Reuses Features to Build Smarter Networks**. 2025. Disponível em: https://medium.com/@caring_smitten_gerbil_914/how-densenet-reuses-features-to-build-smarter-networks-1294f3862498. Acesso em: 06 fev. 2026.
- 30 LONG, Jonathan; SHELHAMER, Evan; DARRELL, Trevor. Fully convolutional networks for semantic segmentation. 7 2015.
- 31 DONG, Chao; LOY, Chen Change; HE, Kaiming; TANG, Xiaoou. Learning a deep convolutional network for image super-resolution. 2014.
- 32 OPENCV. Disponível em: <https://docs.opencv.org/3.4/index.html>. Acesso em: 04 jan. 2026.
- 33 VISUAL STUDIO CODE. **Why did we build Visual Studio Code?** Disponível em: <https://code.visualstudio.com/docs/editor/whyvscode>. Acesso em: 09 mar. 2025.
- 34 CUNHA, Andre o. **React Native: o que é e tudo sobre o Framework**. 2022. Disponível em: <https://www.alura.com.br/artigos/react-native?srsltid=AfmBOorxBH8XvHx02BIPTfNla0McZNo95EnBFQqbTGKSe4JfWncoMhbq>. Acesso em: 12 fev. 2026.
- 35 EXPO. **Expo: Introduction**. Disponível em: <https://docs.expo.dev/get-started/introduction/>. Acesso em: 12 fev. 2026.
- 36 PYTHON. **What is Python? Executive Summary**. Disponível em: <https://www.python.org/doc/essays/blurb/>. Acesso em: 13 fev. 2026.
- 37 SAMSUNG. **Galaxy S20 FE | S20 | S20+ | S20 Ultra LTE**. Disponível em: https://www.samsung.com/africa_pt/smartphones/galaxy-s20/camera/. Acesso em: 30 mar. 2026.

7 Apêndice A - Porção do código em Python para realizar a decodificação e codificação *QR codes*

Código 7.1 – Rotina de manipulação de frames e detecção de códigos bidimensionais

```

1 import websockets
2 import cv2
3 from cv2.wechat_qrcode import WeChatQRCode
4 import qrcode
5 from constants import PHOTO_PATH
6 import os
7 import io
8 import base64
9 import json
10 from datetime import datetime
11 import numpy as np
12 import zxingcpp
13 from src.constants.constants import DETECT_CAFFEMODEL,
    DETECT_PROTOTXT, SR_CAFFEMODEL, SR_PROTOTXT, ECC_MAPPING
14 from qrcode.constants import ERROR_CORRECT_L, ERROR_CORRECT_M,
    ERROR_CORRECT_Q, ERROR_CORRECT_H
15
16 ## Instanciar o objeto Qr code detector
17 try:
18     qrcode_detector = WeChatQRCode(
19         detector_caffe_model_path=DETECT_CAFFEMODEL,
20         detector_prototxt_path=DETECT_PROTOTXT,
21         super_resolution_caffe_model_path=SR_CAFFEMODEL,
22         super_resolution_prototxt_path=SR_PROTOTXT
23     )
24 except Exception as e:
25     print(e)
26     qrcode_detector = None
27
28
29 async def process_image_stream(websocket):

```



```

30     """Gerencia cada conexão de cliente de forma individual"""
31
32     #* Alertar que um novo cliente conectou ao servidor
33     print(f"Novo cliente conectado: {websocket.remote_address}")
34
35     try:
36         #* Loop assíncrono infinito que aguarda novas mensagens
37         #* chegarem do cliente
38         async for msg in websocket:
39             """
40             1. Aqui vai chegar os dados brutos da imagem capturada
41             (em bytes)
42
43             2. É necessário converter os bytes brutos do arrayBuffer
44             em um array numpy de números inteiros
45             para o processamento da imagem
46
47             3. Após converter os bytes brutos em array numpy, é
48             necessário decodificar o JPEG com o cv2
49             """
50
51             #* Converter os bytes brutos em um array numpy de
52             #* números inteiros de 8-bits
53             nparr = np.frombuffer(buffer=msg, dtype=np.uint8)
54
55             #* Decodificar o array numpy
56             frame = cv2.imdecode(nparr, cv2.IMREAD_COLOR)
57
58             #* Verifica se o frame contém informação (caso não
59             #* contenha tudo indica que a informação seja um texto)
60             if frame is not None:
61                 if qrcode_detector:
62                     #* Detectar e decodificar o qr code
63                     res, points =
64                     qrcode_detector.detectAndDecode(frame)
65
66                     #* Verifica se encontrou algum qr code
67                     if len(res) > 0:
68                         qr_info = res[0] #* Pegar a primeira
69                         informação encontrada

```

```

62         qr_points = points[0] ## Obter os pontos da
primeira informação encontrada
63         print("Qr code detectado: ", qr_info)
64
65         ecc_level = get_ecc(
66             decoded_text=qr_info, img_frame=frame
67         )
68
69         img_uri, version, qr_modules =
create_clean_qr_code(
70             info=qr_info, ecc_level_str=ecc_level
71         )
72
73         pts = qr_points.astype(int)
74         x,y,w,h = cv2.boundingRect(pts)
75         pixel_size = f"{w}x{h}"
76         # timestamp =
datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
77         # filepath = os.path.join(PHOTO_PATH,
f"{timestamp}_{version}_{ecc_level}.png")
78         # success = cv2.imwrite(filename=filepath,
img=frame)
79
80         await websocket.send(json.dumps({
81             "type": "detection_result",
82             "content_info": qr_info,
83             "image_uri": img_uri,
84             "metadata": {
85                 "version": str(version), # Esta é a
versão do QR Code GERADO (Limpo)
86                 "ecc": ECC_MAPPING.get(ecc_level,
'Desconhecido'), # ECC do ORIGINAL
87                 "pixel_size": pixel_size,
88                 "module":
f"{qr_modules}x{qr_modules}",
89                 "size_bytes":
len(qr_info.encode('utf-8'))
90             }
91         })))
92         else:

```

```

93         ## Verificar se é um texto para a geração de um novo
QR code
94         try:
95             ## Decodificar os bytes para string do tipo UTF-8
96             decoded_text = msg.decode('utf-8')
97
98             print(decoded_text)
99
100            print(f"[Gerador] Criando QR para:
101            {decoded_text}")
102
103            ## Gerar um novo QR Code limpo
104            img_uri, version, qr_modules =
105            create_clean_qr_code(
106                info=decoded_text
107            )
108
109            ## Envia a informação codificada ao usuário
110            await websocket.send(json.dumps({
111                "type": "code_generation",
112                "content_info": decoded_text,
113                "image_uri": img_uri,
114                "metadata": {
115                    "version": version,
116                    "pixel_size":
117                    get_qr_code_size(info=decoded_text),
118                    "module": f"{qr_modules}x{qr_modules}",
119                    "size_bytes":
120                    len(decoded_text.encode('utf-8'))
121                }
122            }))
123
124            except UnicodeDecodeError:
125                print(
126                    "Recebido dados desconhecidos ou corrompidos
127                    ao decodificar o texto."
128                )
129
130            except Exception as e:
131                print("Erro ao gerar o código: ", e)
132
133            except websockets.exceptions.ConnectionClosed:
134                print("O cliente desconectou do servidor")
135
136            except Exception as e:

```

```
128         print(f"Erro inesperado: {e}")
```

Fonte: Autor.