Fernando Martins Pereira Neto

APLICATIVO PARA RECONHECIMENTO DE CARACTERES JAPONESES ATRAVÉS DE DEEP LEARNING

Monografia apresentada ao programa de Graduação em Engenharia de Informação da Universidade Federal do ABC (UFABC), como requisito parcial à obtenção do título de Bacharel em Engenharia de Informação.

Orientador: Prof. Dr. Kenji Nose Filho

Santo André - SP
2020

RESUMO

Os *Kanjis* mais comuns representam uma lista de mais de 2000 símbolos no sistema de escrita japonês. A ideia do projeto é desenvolver um aplicativo de celular que auxilie estudantes da língua japonesa a reconhecerem estes símbolos determinando suas pronúncias e significados. Para isso, foi treinado um classificador com *Deep Learning* em rede neural convolucional.

Para treinar o classificador, é preciso primeiro ter um número grande de imagens contendo estes símbolos para usar como amostras no treinamento. Para isso, foram usadas fontes de escrita japonesas e extraídas suas imagens usando um editor de texto, além da aplicação de *data augmentation* para aumentar a variedade de amostras.

Com o banco de imagens terminado, foi feita a elaboração da rede neural convolucional em Keras, que permite a estruturação de redes neurais em poucas linhas de código. Após vários testes e ajustes finos de parâmetros, foi possível criar um reconhecedor com 90% de acurácia, valor adequado para o problema em questão contendo mais de 2000 classes.

O *framework* para o desenvolvimento do aplicativo foi o Flutter, *kit* de desenvolvimento de aplicativos na linguagem Dart. O aplicativo final ficou dividido em duas páginas. A tela principal contém o canvas para o usuário desenhar o *Kanji* a ser reconhecido, que após o desenho é gerada em tempo real uma lista de possíveis *Kanjis* previstos pela rede neural treinada. Com isso, o usuário pode clicar no símbolo desejado, levando o *app* à tela seguinte contendo informações como número de traços, significado e pronúncia.

SUMÁRIO

1 - INTRODUÇÃO

Um dos maiores obstáculos no aprendizado da língua japonesa é a memorização dos mais de 2000 símbolos usados comumente na linguagem. Eles estão separados em três categorias: *Hiragana*, *Katakana* e *Kanji*. O *Hiragana* é usado para várias palavras de origem japonesa e de papel semântico, como as partículas (similares às preposições). O *Katakana* é usado para termos de origem estrangeira, geralmente vindos do inglês. E os *Kanjis* são símbolos vindos do sistema linguístico chinês que possuem significado próprio.

ん ン n	わ ワ wa	ら ラ ra	や ヤ ya	ま マ ma	は ハ ha	な ナ na	た タ ta	さ サ sa	か カ ka	あ ア a
		り リ ri		み ミ mi	ひ ヒ hi	に ニ ni	ち チ chi	し シ shi	き キ ki	い イ i
		る ル ru	ゆ ユ yu	む ム mu	ふ フ fu	ぬ ヌ nu	つ ツ tsu	す ス su	く ク ku	う ウ u
		れ レ re		め メ me	へ ヘ he	ね ネ ne	て テ te	せ セ se	け ケ ke	え エ e
	を ヲ wo	ろ ロ ro	よ ヨ yo	も モ mo	ほ ホ ho	の ノ no	と ト to	そ ソ so	こ コ ko	お オ o

Figura 1 - Tabela[1] para os *kanas* (Hiragana e Katakana).

Na tabela da Figura 1, o *Hiragana* está à esquerda de cada quadrante e em geral tem traços mais curvilíneos, enquanto o *Katakana* à direita possui traços mais retos. Existem ainda algumas junções que alteram pronúncia envolvendo a coluna *ya-yu-yo*, além de símbolos pontuais para gerar a pronúncia de algumas consoantes não presentes na tabela, mas os símbolos base dos *kanas* são estes 92 símbolos (46 cada). Com a ajuda de mnemônicos, é possível memorizá-los superficialmente em poucas semanas.

A maior dificuldade é aprender os símbolos da terceira categoria, os *Kanjis*. Os *jouyou kanji*, literalmente "caracteres chineses de uso regular", fazem parte de uma lista de 2136 símbolos anunciados oficialmente pelo Ministro da Educação do Japão, número atualizado pela última vez no ano de 2010 (WIKIPEDIA, 2020). É uma lista geral que, apesar de ainda ser possível encontrar algum símbolo fora dessa lista em textos japoneses, representa os mais frequentes.

Uma outra dificuldade é o fato de que cada *Kanji* possui significados específicos e normalmente mais de uma pronúncia diferente, enquanto os *kanas* não têm significado próprio e possuem uma única pronúncia. A lista completa dos

[1] Disponível em https://www.reddit.com/r/LearnJapanese/comments/b0jlyt/i_made_an_allinone_hiraganakatakanabasic_kanji/. Acessado em 27 out. 2020.

jouyou kanji pode ser vista na Figura 2, e na Figura 3 é possível ver de quantos *Kanjis* um único parágrafo pode ser composto.

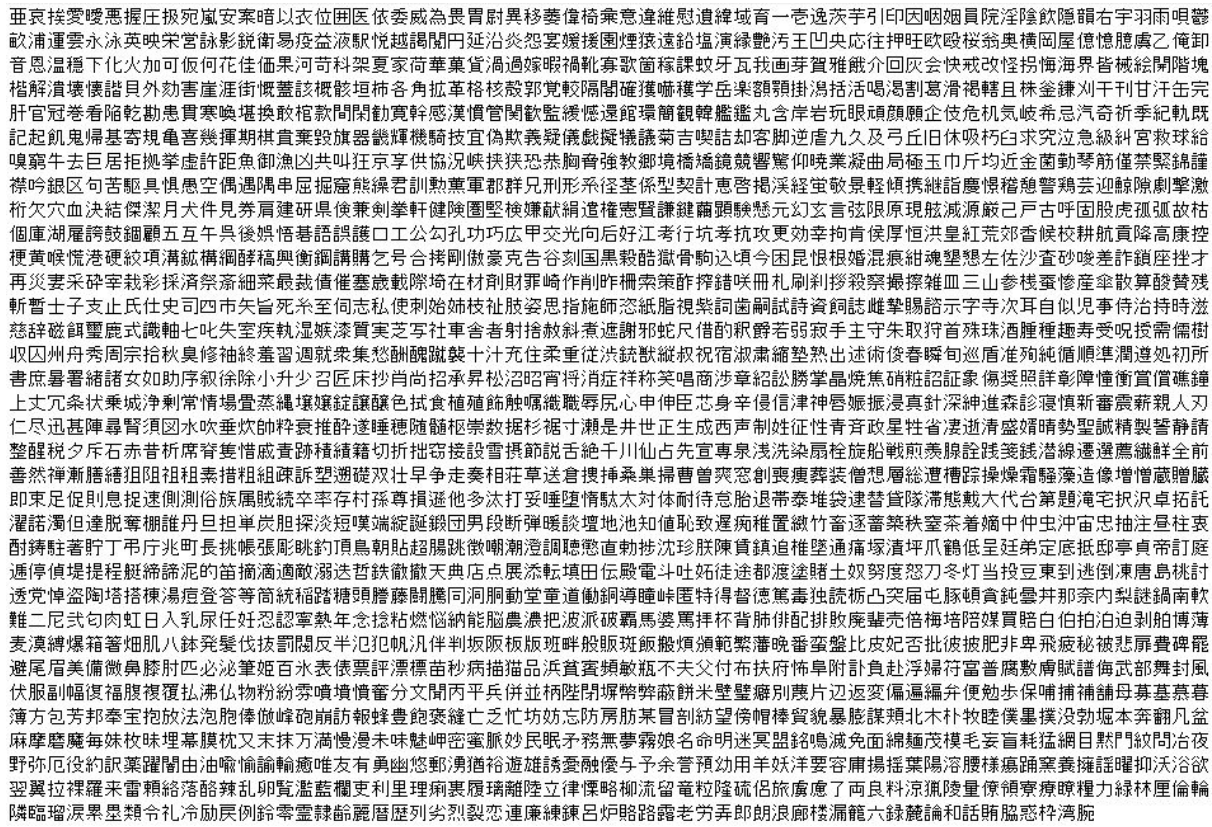


Figura 2 - Todos os 2136 *jouyou kanji*.

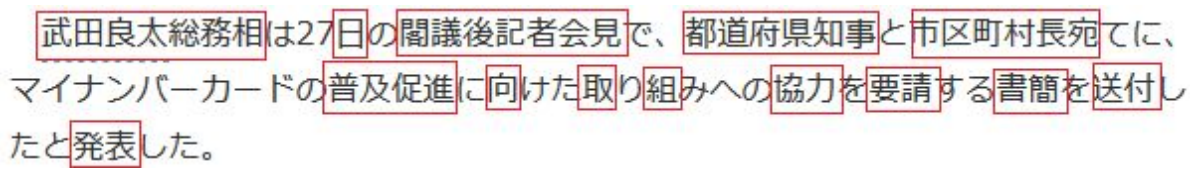


Figura 3 - *Kanjis* presentes em um parágrafo de uma notícia do site Yahoo JP.

Especialmente para estudantes estrangeiros do japonês, existem vários *softwares* e aplicativos de celular que auxiliam na identificação, leitura e tradução destes símbolos, sejam dicionários bilíngues ou reconhecedores ópticos de símbolos. A ideia deste projeto é criar um aplicativo de celular gratuito que faça o reconhecimento de um único *Kanji* que o usuário desenhe, mostrando seu significado e pronúncias, já que os *apps* existentes que fazem tarefa parecida ou são pagos, ou precisam levar em consideração a ordem dos traços ou reconhecem apenas fotos e/ou palavras completas sem informação específica do *Kanji*.

O método escolhido para fazer o reconhecimento dos *Kanjis* foi por tratamento de arquivo de imagem via *Deep Learning* estruturado em uma rede neural convolucional. Ou seja, a partir de um banco de imagens de todos os 2136 *jouyou kanji*, foi estruturado um aprendizado de máquina usando estas imagens como dados de entrada do treinamento.

Deep Learning é um ramo de *machine learning* que usa grafos profundos com várias camadas de processamento interligadas, substituindo a necessidade de programar manualmente a aquisição das características mais importantes do dado a se tratar (o *feature engineering*). A rede neural convolucional é uma classe de rede neural artificial que vem tendo comprovada sua eficácia no processamento de imagens. Estes conceitos serão aprofundados mais adiante.

O projeto ficou então separado em três etapas: aquisição do banco de imagens, treinamento e produção do reconhecedor de *Kanjis* e desenvolvimento do aplicativo de celular com base no reconhecedor.

2 - AQUISIÇÃO DO BANCO DE IMAGENS

A ideia é utilizar um número grande de imagens diferentes de cada um dos 2136 *Kanjis* para o treinamento devolver uma acurácia adequada. Para isso, o primeiro passo é adquirir várias fontes japonesas diferentes (como a *Times New Roman* e a *Arial*, estas ocidentais) e utilizá-las num editor de texto para posterior exportação em imagem.

Foi possível adquirir gratuitamente 35 fontes do site *Free Japanese Fonts*[2]. De lá foram baixados os arquivos **.ttf** e **.otf**, que instalam e deixam visíveis as fontes como opção no campo de fontes no editor de texto (no caso do projeto, o Libre Office Writer). Elas incluem tanto fontes padrões encontradas em livros e documentos como também incluem algumas desenhadas a mão por pessoas diferentes, além de todas possuírem variação de estilo e largura de traço, de acordo com a Figura 4.

Figura 4 - *Kanji* 嵐 (*arashi*, tempestade) nas 35 fontes.

Para utilizá-las no editor de texto, foi necessário primeiro adquirir a lista de todos os *jouyou kanji* e já definir uma indexação do *Kanji* 1 ao *Kanji* 2136. Foi

[2] Disponível em https://www.freejapanesefont.com/. Acessado em 27 out. 2020.

decidido seguir a ordem padrão mostrada no site *Kanji Database*[3], que basicamente mantém a ordem de pronúncia a-i-u-e-o da direita para a esquerda mostrada na tabela dos *kanas*.

Com as fontes e a lista dos *Kanjis* em mãos, foi possível gerar um arquivo **.doc** para cada uma das 35 fontes contendo os 2136 símbolos. Por experimentação para deixar num tamanho adequado, foi decidido deixar os caracteres num tamanho em que coubessem 17 por fileira na página.

Porém, cada fonte possuia margens diferentes entre suas linhas (alturas diferentes), não deixando constante a quantidade de símbolos por coluna na página (na primeira coube 19 por coluna, na segunda 21, etc). A quantidade de símbolos por coluna na página foi anotada para cada fonte para ser utilizada no processo de segmentação. Além disso, foi necessário ajustar manualmente as margens nas extremidades para que ficassem num posicionamento simétrico. Um exemplo desta diferença entre margens verticais pode ser visto na Figura 5.

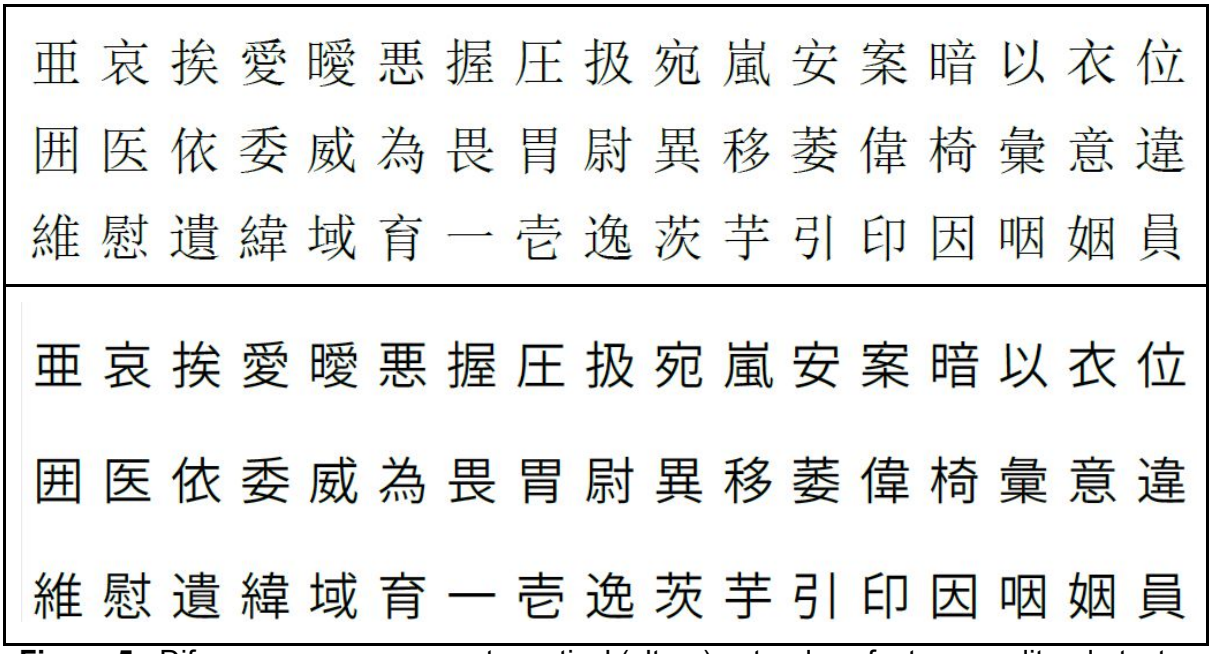


Figura 5 - Diferença no espaçamento vertical (altura) entre duas fontes no editor de texto.

Após esta tarefa, os .docs foram convertidos em **PDF**, que por sua vez foram convertidos em **.jpg** para serem usados na automatização da segmentação.

Foi decidido usar a linguagem Python para fazer o recorte de cada *Kanji* a partir das **JPGs**. Ela possui a biblioteca Pandas que permite tratar e armazenar dados tabulares eficientemente a partir de um objeto `DataFrame`, onde cada linha foi padronizada para ser o índice do *Kanji* de 1 a 2136, e inicialmente com 35 colunas, uma para cada fonte. Por exemplo, a célula da linha 2 e coluna 5 contém a matriz de pixels da imagem para o *Kanji* de índice 2 e a quinta fonte.

Para o processamento das imagens, foi usada a biblioteca cv2 (CV = *computer vision*), que é a implementação do OpenCV para Python. O ambiente de

[3] Disponível em https://www.kanjidatabase.com/kanji_database.php. Acessado em 27 out. 2020.

desenvolvimento escolhido foi o Jupyter, que usa *notebooks* **.ipynb** permitindo o desenvolvimento separado em blocos (*shell*).

```
import pandas as pd
import numpy as np

import cv2
import matplotlib.pyplot as plt
```

1) Segmentation from full pages

```
# 35 fonts; list of 35 values
font_kanjispercol = [19, 21, 21, 14, 21, 14, 14,
                     21, 21, 15, 21, 14, 21, 13,
                     21, 21, 21, 21, 21, 21, 21,
                     21, 21, 21, 14, 14, 21, 21,
                     20, 21, 21, 21, 21, 21, 21]
font_kanjisperrow = 17 # constant for all 35 fonts
total_kanjis = 2136
binary_threshold = round(256*0.7) - 1

df_kanjis = pd.DataFrame(columns=[i+1 for i in range(total_kanjis)]).T

for font in [i+1 for i in range(len(font_kanjispercol))]:
    kanjis_col = font_kanjispercol[font-1]
    kanjis_row = font_kanjisperrow
    kanjis_per_page = kanjis_row*kanjis_col
    pages = int(total_kanjis / kanjis_per_page) + 1
    index = 1
    print('Starting font %s...'%font)

    for page in [i+1 for i in range(pages)]:
        im_path = 'C:/Users/ferna/Desktop/tg/git-KanjiOCR_Sampling/jpgs/%s/%s-%s.jpg' % (font, font, page)
        im_gray = cv2.imread(im_path, cv2.IMREAD_GRAYSCALE)
        # Convert to binary
        im = cv2.threshold(im_gray, binary_threshold, 255, cv2.THRESH_BINARY)[1]
        # Adjust higher threshold for font 11 (too thin)
        if font==11: im = cv2.threshold(im_gray, 222, 255, cv2.THRESH_BINARY)[1]
        height = im.shape[0]
        width = im.shape[1]
        kanji_height = round(height/kanjis_col)
        kanji_width = round(width/kanjis_row)
        # Get more symetric starting points
        starting_row = round((height - kanjis_col*kanji_height)/2)
        if starting_row < 0: starting_row = 0
        starting_col = round((width - kanjis_row*kanji_width)/2)
        if starting_col < 0: starting_col = 0

        # Segment according to kanjis por row/col
        for row in range(kanjis_col):
            for col in range(kanjis_row):
                filepath = 'C:/Users/ferna/Desktop/test/%s/%s.png' % (font, index)
                r0 = starting_row + row*kanji_height
                r1 = starting_row + (row+1)*kanji_height
                c0 = starting_col + col*kanji_width
                c1 = starting_col + (col+1)*kanji_width
                cv2.imwrite(filepath, im[r0:r1, c0:c1])
                df_kanjis.loc[index, font] = ''
                df_kanjis.loc[index, font] = im[r0:r1, c0:c1]
                index = index + 1
                if index > total_kanjis: break
            if index > total_kanjis: break
        if index > total_kanjis: break
df_kanjis.to_pickle('df_seg.pkl')
```

2135	[[255, 255, 255, 255, 255, 255, 255, 255, 255,...	[[255, 255, 255, 255, 255, 255, 255, 255, 255,...	[[255, 255, 255, 255, 255, 255, 255, 255, 255,...	[[255, 255, 255, 255, 255, 255, 255, 255, 255,...	[[255, 255, 255, 255, 255, 255, 255, 255, 255,...	[[255, 255, 255, 255, 255, 255, 255, 255, 255,...	[[255, 255, 255, 255, 255, 255, 255, 255, 255,...	[[255, 255, 255, 255, 255, 255, 255, 255, 255,...	[[255, 255, 255, 255, 255, 255, 255, 255, 255,...	[[255, 255, 255, 255, 255, 255, 255, 255, 255,...	...	[[255, 255, 255, 255, 255, 255, 255, 255, 255,...	[[255, 255, 255, 255, 255, 255, 255, 255, 255,...	[[255, 255, 255, 255, 255, 255, 255, 255, 255,...	[[255, 255, 255, 255, 255, 255, 255, 255, 255,...	[[255, 255, 255, 255, 255, 255, 255, 255, 255,...	[[255, 255, 255, 255, 255, 255, 255, 255, 255,...	[[255, 255, 255, 255, 255, 255, 255, 255, 255,...	[[255, 255, 255, 255, 255, 255, 255, 255, 255,...	[[255, 255, 255, 255, 255, 255, 255, 255, 255,...	[[255, 255, 255, 255, 255, 255, 255, 255, 255,...
2136	[[255, 255, 255, 255, 255, 255, 255, 255, 255,...	[[255, 255, 255, 255, 255, 255, 255, 255, 255,...	[[255, 255, 255, 255, 255, 255, 255, 255, 255,...	[[255, 255, 255, 255, 255, 255, 255, 255, 255,...	[[255, 255, 255, 255, 255, 255, 255, 255, 255,...	[[255, 255, 255, 255, 255, 255, 255, 255, 255,...	[[255, 255, 255, 255, 255, 255, 255, 255, 255,...	[[255, 255, 255, 255, 255, 255, 255, 255, 255,...	[[255, 255, 255, 255, 255, 255, 255, 255, 255,...	[[255, 255, 255, 255, 255, 255, 255, 255, 255,...	...	[[255, 255, 255, 255, 255, 255, 255, 255, 255,...	[[255, 255, 255, 255, 255, 255, 255, 255, 255,...	[[255, 255, 255, 255, 255, 255, 255, 255, 255,...	[[255, 255, 255, 255, 255, 255, 255, 255, 255,...	[[255, 255, 255, 255, 255, 255, 255, 255, 255,...	[[255, 255, 255, 255, 255, 255, 255, 255, 255,...	[[255, 255, 255, 255, 255, 255, 255, 255, 255,...	[[255, 255, 255, 255, 255, 255, 255, 255, 255,...	[[255, 255, 255, 255, 255, 255, 255, 255, 255,...	[[255, 255, 255, 255, 255, 255, 255, 255, 255,...

2136 rows × 35 columns

Figura 6 - Código para segmentação a partir das **JPGs**, e preview do `DataFrame` com as matrizes de imagem.

O que o código da Figura 6 basicamente faz é descobrir as dimensões em pixels da **JPG** e cortar simetricamente de acordo com o número de linhas e colunas na página, usando os valores da lista `font_kanjispercol` e sabendo que cada linha sempre possui 17 *Kanjis*. Foram feitos 4 loops aninhados para iterar sobre cada fonte, cada página da fonte, cada linha da página e cada coluna da linha. Foi usada a biblioteca cv2 para ler as imagens e binarizá-las com um limiar arbitrário, fazendo com que os pixels só possuam valor 0 (preto) e 255 (branco).

Por fim, o `DataFrame` contendo as matrizes de pixel das imagens foi salvo no formato **.pkl** (***pickle***), que permite armazenamento e leitura eficientes deste tipo de objeto.

Como visto no *preview* do `DataFrame`, as imagens todas estão com espaço em branco envoltório (pixels com valor 255). A seguir, foi feito o *crop* deste envoltório com uma função (definida na Figura 7) que verificava se todos os valores da linha ou coluna eram 255, e com um *loop* para cada uma das 4 direções foi possível definir onde não havia mais linhas em branco e cortar as excedentes.

Outra vantagem de se usar `DataFrame` neste caso é poder usar seu método `applymap` passando uma função como parâmetro, já que este método aplica a função em todos os membros do `DataFrame` sem precisar estruturar laços. Portanto, bastou apenas recarregar o arquivo ***pickle***, definir a função e aplicá-la usando tal método.

```
10 def remove_whitespace(sample):
11     height = len(sample[:,0])
12     width = len(sample[0,:])
13     whiterow_sum = width*255
14     whitecol_sum = height*255
15     # Stop at rows/columns where there's at least 1 black pixel
16     for row1 in range(height):
17         if sample[row1,:].sum()!=whiterow_sum: break
18     for row2 in reversed(range(height)):
19         if sample[row2,:].sum()!=whiterow_sum: break
20     for col1 in range(width):
21         if sample[:,col1].sum()!=whitecol_sum: break
22     for col2 in reversed(range(width)):
23         if sample[:,col2].sum()!=whitecol_sum: break
24     return sample[row1:row2+1,col1:col2+1]
```

Figura 7 - Função que remove o espaço em branco envoltório de uma matriz de pixels.

Com isso, tem-se 35 imagens para cada um dos 2136 caracteres, totalizando 74760 imagens. Este número é muito inferior em relação à quantidade necessária para se treinar um classificador com 2136 classes diferentes. É preciso ampliar consideravelmente o número de imagens para o treinamento.

Uma abordagem que resolve este problema chama-se *data augmentation*, termo que significa aumentar a variedade de dados de entrada sem precisar coletar dados novos. Em imagens, isso é possível por meio da aplicação de transformações

lineares e distorções, mantendo os arquivos originais e criando novos arquivos transformados.

Os quatro tipos de transformações que foram utilizadas são rotações, perspectiva horizontal, perspectiva vertical e taxas de compressão. A metodologia é quase a mesma do crop do envoltório branco: definir uma função que recebe uma imagem e faz a transformação desejada, e aplicá-la no `DataFrame` com o método `applymap`. Só que desta vez não foram sobrescritas as imagens anteriores, havendo a concatenação de `DataFrames`. Além disso, estas funções agora recebem parâmetros além da própria imagem, para facilitar a aplicação de transformações em magnitudes diferentes.

```
def rotate_image(image, angle):
    image_center = tuple(np.array(image.shape[1::-1]) / 2)
    rot_mat = cv2.getRotationMatrix2D(image_center, angle, 1.0)
    result = cv2.warpAffine(image, rot_mat, image.shape[1::-1], flags=cv2.INTER_LINEAR)
    return result
```

```
def rotate_image_lossless(image, angle):
    # Get difference of height/width of rotation
    half_height = image.shape[0]/2
    half_width = image.shape[1]/2
    half_diagonal = (half_height**2 + half_width**2)**0.5
    diag_angle = np.arctan(half_height/half_width)
    angle_rad = np.deg2rad(angle)
    diff_height = half_diagonal*(np.sin(diag_angle + angle_rad) - np.sin(diag_angle))
    diff_height = int(abs(diff_height)) + 1
    diff_width = half_diagonal*(np.cos(diag_angle + angle_rad) - np.cos(diag_angle))
    diff_width = int(abs(diff_width)) + 1

    # Pad image before rotating so there's no unwanted crop
    inverted_im = 255 - image
    image_pad = cv2.copyMakeBorder(inverted_im, diff_height, diff_height,
                                   diff_width, diff_width, cv2.BORDER_CONSTANT)
    # Rotate, invert colors back to normal and binarize again
    final_im = 255 - rotate_image(image_pad, angle)
    final_im = cv2.threshold(final_im, 0, 255, cv2.THRESH_BINARY | cv2.THRESH_OTSU)[1]
    final_im = remove_whitespace(final_im)
    return final_im
```

```
df_rotated = df_cropped.copy()
for angle in list_of_angles:
    print('Rotating %s degrees...' % angle)
    df_temp = df_cropped.applymap(lambda x: rotate_image_lossless(x, angle))
    df_rotated = pd.concat([df_rotated, df_temp], axis=1)
    del df_temp

number_of_columns = len(df_rotated.columns)
df_rotated.columns = [i+1 for i in range(number_of_columns)]
del df_cropped
df_rotated.to_pickle('df_rotated.pkl')
del df_rotated
```

Figura 8 - Funções para rotação da imagem e sua aplicação com uma lista de ângulos.

Seguindo o código da Figura 8, inicia-se com a transformação de rotação. Decidiu-se por aplicar 4 rotações diferentes indicadas no *array* `list_of_angles`,

de valor [-4, -2, 2, 4]. A função auxiliar `rotate_image` aplica a transformação geométrica *affine* a partir de uma matriz de rotação 2D, rotacionando a imagem no ângulo desejado. O problema é que ela mantém as dimensões originais pré-transformação, cortando parte da imagem, além de manter o fundo preto como resquício da posição anterior em vez de manter em branco.

Então, na função `rotate_image_lossless`, são calculadas as diferenças de posição da meia-diagonal da imagem pré e pós rotação, para definir quantos pixels são necessários adicionar no *padding* da largura e comprimento. Primeiro, inverte-se a imagem (255 subtraindo os valores dos pixels) para deixar o fundo preto, e depois aplica-se o *padding* com a função `cv2.copyMakeBorder`, para em seguida aplicar a função `rotate_image` e reverter para fundo branco, seguida de uma nova binarização para eliminar os pixels intermediários gerados na rotação e finalizando com a remoção do envoltório branco excedente.

Fazendo um loop sobre a lista contendo os ângulos -4º, -2º, 2º e 4º, obtém-se 140 (35x4) novas imagens, que somadas às 35 iniciais totalizam 175 imagens por *Kanji*. Terminado o *data augmentation* por rotação, partiu-se para a adição de perspectiva.

```
def add_perspective(sample, direction, percentage):
    inverted_im = 255 - sample
    h1, w1 = inverted_im.shape
    # 4 extremity points from original image
    pts1 = [[0, 0], [w1-1, 0], [0, h1-1], [w1-1, h1-1]]
    # 4 extremity points, where 2 are distorted according to direction
    if direction=='left_to_right':
        dist = round(h1*percentage/2)
        pts2 = [[0, 0], [w1-1, dist], [0, h1-1], [w1-1, h1-1-dist]]
    elif direction=='right_to_left':
        dist = round(h1*percentage/2)
        pts2 = [[0, dist], [w1-1, 0], [0, h1-1-dist], [w1-1, h1-1]]
    elif direction=='top_down':
        dist = round(w1*percentage/2)
        pts2 = [[0, 0], [w1-1, 0], [dist, h1-1], [w1-1-dist, h1-1]]
    elif direction=='bottom_up':
        dist = round(w1*percentage/2)
        pts2 = [[dist, 0], [w1-1-dist, 0], [0, h1-1], [w1-1, h1-1]]
    pts1, pts2 = (np.array(pts1), np.array(pts2))

    h, mask = cv2.findHomography(pts1, pts2, cv2.RANSAC)
    im_perspec = cv2.warpPerspective(inverted_im, h, (w1, h1))
    final_im = 255 - im_perspec
    final_im = cv2.threshold(final_im, 0, 255, cv2.THRESH_BINARY | cv2.THRESH_OTSU)[1]
    final_im = remove_whitespace(final_im)
    return final_im
```

Figura 9 - Função que recebe a imagem, direção da perspectiva e porcentagem de perspectiva a ser aplicada.

A função `add_perspective` (Figura 9) começa definindo os pontos para os 4 cantos da imagem. Com a direção (esquerda para a direita ou vice-versa ou de cima para baixo ou vice-versa) e porcentagem de distorção dadas, são calculados novos 4 pontos para os cantos da imagem. Por exemplo, para aplicar uma distorção

da esquerda para a direita e 50% de distorção, para uma imagem 4x4 pixels de cantos com coordenadas (1, 1), (4, 1), (1, 4) e (4, 4), os novos 4 cantos terão coordenadas (1, 1), (4, 2), (1, 4) e (4, 3). O exemplo foi feito considerando coordenadas iniciando em 1 para facilitar a elucidação, porém coordenadas na linguagem iniciam em 0.

Com os novos 4 cantos em mãos, é aplicada a projeção da imagem a partir das funções `cv2.findHomography` e `cv2.warpPerspective`. Como na transformação por rotação, foi preciso tomar cuidado com os resquícios de cor preta e aplicar novamente a remoção do envoltório branco.

Definida a função, partiu-se primeiro para a adição de perspectiva horizontal, da esquerda para a direita e da direita para a esquerda, com 20% de distorção. Aplicando as duas distorções sobre as 175 imagens anteriores, tem-se no total 525 imagens por *Kanji*. Fazendo o mesmo com distorções de cima para baixo e de baixo para cima na mesma porcentagem sobre as 525 imagens, foram obtidas 1575 imagens por *Kanji*.

Vale notar que tudo isto está sendo feito com o método `applymap` do objeto `DataFrame`, que apesar de nesse caso ser consideravelmente mais rápido que loops tradicionais, tem alto custo de memória RAM. O projeto foi feito em um *notebook* com 8 GB de memória RAM, e a partir desta etapa de perspectivas horizontal e vertical, foi preciso carregar os `DataFrames` divididos em partes (*chunks*). Então, se após as rotações foi gerado apenas 1 arquivo ***pickle***, foi preciso dividir em 4 arquivos as imagens com perspectiva horizontal (primeiro bloco de código da Figura 10, seguido pela aplicação da função no segundo bloco) e em 12 arquivos as imagens com perspectiva vertical, para já aliviar a memória na transformação final.

```
# Divide in chunks to preserve memory
chunk_cols1 = [i for i in range(1,45)] #1 to 44
chunk_cols2 = [i for i in range(45,89)] #45 to 88
chunk_cols3 = [i for i in range(89,133)] #89 to 132
chunk_cols4 = [i for i in range(133,176)] #133 to 175
list_of_chunks = [chunk_cols1, chunk_cols2, chunk_cols3, chunk_cols4]
```

```
percentage = 0.2
file = 1
for chunk in list_of_chunks:
    print('####################################################')
    df_rotated = pd.read_pickle('df_rotated.pkl')[chunk]
    print('Read chunk %s.' % file)
    df_horizontal = df_rotated.copy()
    for direction in ['left_to_right', 'right_to_left']:
        print('Started ' + direction)
        df_temp = df_rotated.applymap(lambda x: add_perspective(x, direction, percentage))
        print('Finished ' + direction)
        df_horizontal = pd.concat([df_horizontal, df_temp], axis=1)
        print('Concat done.')
        del df_temp

    number_of_columns = len(df_horizontal.columns)
    df_horizontal.columns = [i+1 for i in range(number_of_columns)]
    del df_rotated
    df_horizontal.to_pickle('df_horizontal%s.pkl' % file)
    del df_horizontal
    print('Finished chunk %s.' % file)
    file = file + 1
```

Figura 10 - Divisão em chunks e aplicação da perspectiva horizontal para 4 arquivos.

O último passo do *data augmentation* é aplicar taxas de compressão diferentes, ao mesmo tempo em que se define as dimensões padronizadas para a imagem final, que ficou decidido como 48x48 pixels, resolução suficiente para manter claros os detalhes dos traços.

```
list_of_compressions = [1, 0.7, 0.4]
new_size = (48,48)

def compress_and_resize(sample, compression_perc, new_size):
    # First, resize it according to compression percentage; then resize it to wanted final size
    resized_shape = tuple(round(i*compression_perc) for i in sample.shape)
    img = cv2.resize(sample, resized_shape)
    img = cv2.resize(img, new_size)
    img = cv2.threshold(img, 0, 255, cv2.THRESH_BINARY | cv2.THRESH_OTSU)[1]
    return img
```

Figura 11 - Porcentagens de compressões, tamanho padrão final e função para aplicação das compressões.

Como indicado na Figura 11, a função de compressão ficou mais simples do que as anteriores, bastando apenas usar a função `cv2.resize` e reaplicar a binarização para remover pixels intermediários. Foi passado também na lista de porcentagens de compressão o valor 100%, para no mesmo passo redimensionar as imagens do ***pickle*** anterior para 48x48 pixels. No final, foram obtidas 1575x3 = 4725 imagens por *Kanji*, totalizando 10,092600 milhões de imagens, um número mais apropriado para usar na etapa de treinamento. Um exemplo de imagens geradas no *data augmentation* para uma única imagem original pode ser visto na Figura 12.

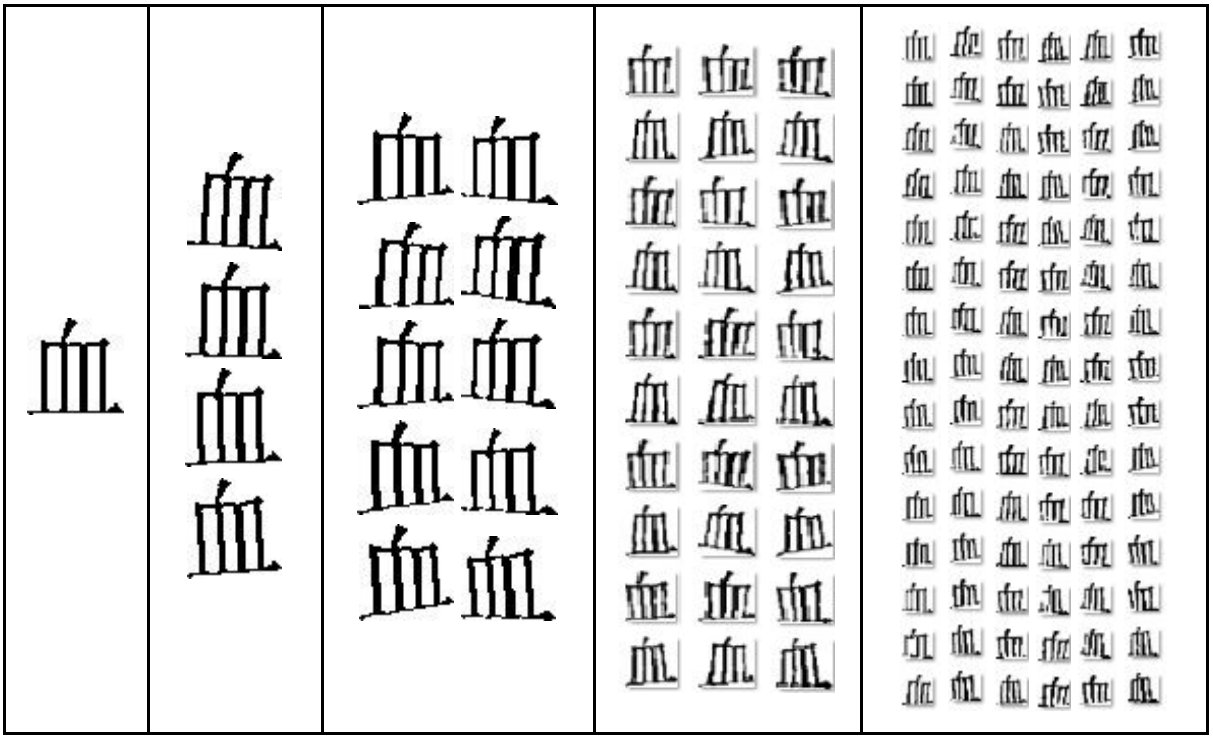


Figura 12 - 1 imagem original, 4 imagens rotacionadas, 10 imagens distorcidas horizontalmente, 30 imagens distorcidas verticalmente, e 90 comprimidas, totalizando 135 para uma única imagem original.

3 - TREINAMENTO E PRODUÇÃO DO RECONHECEDOR DE KANJIS

A próxima etapa é o treinamento através de Deep Learning, modelando o problema com redes neurais convolucionais (*Convolutional Neural Networks*, CNNs). *Deep Learning* extrai e pondera características das imagens de maneira inconsciente (sem necessidade de engenharia de *features*), e CNNs em específico têm uma arquitetura que simula a formação e classificação de imagens de um cérebro humano, onde neurônios recebem os impulsos recebidos pela imagem formada na retina, estruturados em camadas e ligados espacialmente de maneira local (DESHPANDE, 2016).

Uma rede neural convolucional pode ser separada em três tipos de camadas: camada de entrada, onde é recebido o *input* da imagem (matriz dos pixels); camadas
ocultas, onde é feita a ponderação dos pesos a partir de funções de ativação, que extraem características lineares e não-lineares; e a camada de saída, onde é feita a classificação final. O *layout* mais simples de uma rede neural profunda pode ser visto na Figura 13.

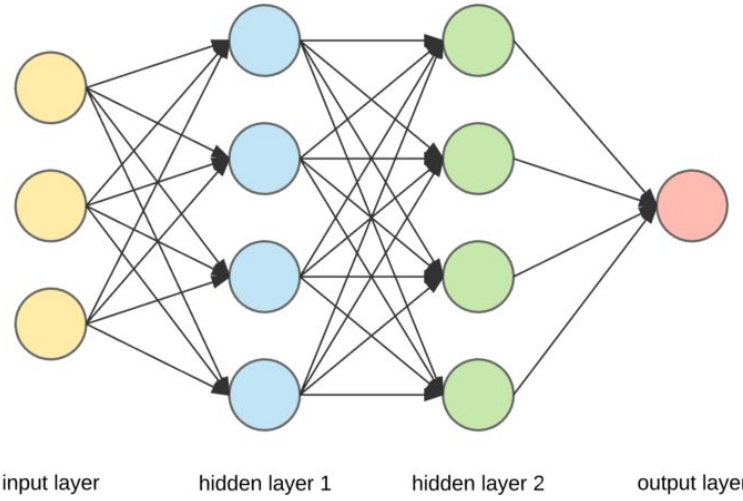


Figura 13 - *Layout* simplificado de uma rede neural multicamadas (mais de uma camada oculta)[4].

O formato para os valores de entrada são os valores dos pixels da matriz da imagem binarizada. Cada imagem de entrada tem uma saída esperada associada. Por exemplo, se a imagem é o *Kanji* 月 (*getsu*, lua), e ele estiver organizado como o segundo da lista, a saída esperada deve estar na forma *Y = [0 1 0 0 0 ... 0]*, onde cada termo do vetor indica a probabilidade da classificação estar correta para cada *Kanji* da lista.

Na primeira camada oculta, normalmente uma camada convolucional, cada neurônio recebe uma fração do total de pixels da entrada de acordo com uma janela (de dimensão 5x5, por exemplo), e a saída de cada neurônio é a soma da multiplicação dos pixels pelos pesos associados a cada entrada. Os valores iniciais dos pesos podem ser aleatórios ou arbitrários, já que o propósito do treinamento é justamente fazer o ajuste destes pesos (NIELSEN, 2015).

A segunda camada oculta pode ser uma camada de ativação, onde são recebidos os valores da camada anterior para serem introduzidas não-linearidades, por exemplo através da função Unidade Linear Retificada (ReLU), definida por *f(x) = max(0,x)*. Basicamente, usa-se esta função para aumentar propriedades não-lineares com pouco gasto computacional e sem acarretar desvanecimento dos valores.

A terceira camada oculta é a camada de *pooling*, onde é feito basicamente um *downsampling*. É aplicada uma janela, normalmente de tamanho 2x2, e extraído o máximo de cada janela. Isto é feito para diminuir consideravelmente o número de parâmetros e, com isso, custo computacional, e também para aumentar a capacidade de generalização do processo, diminuindo efeitos de *overfitting* (quando a acurácia é alta no reconhecimento de dados usados no treinamento e consideravelmente mais baixa em dados não vistos). A ordem e a quantidade de camadas ocultas podem variar, e um exemplo de CNN pode ser visto na Figura 14.

[4] Figura extraída de "Applied Deep Learning - Part 1: Artificial Neural Networks | by Arden Dertat" https://towardsdatascience.com/applied-deep-learning-part-1-artificial-neural-networks-d7834f67a4f6. Acessado em 31 out. 2020.

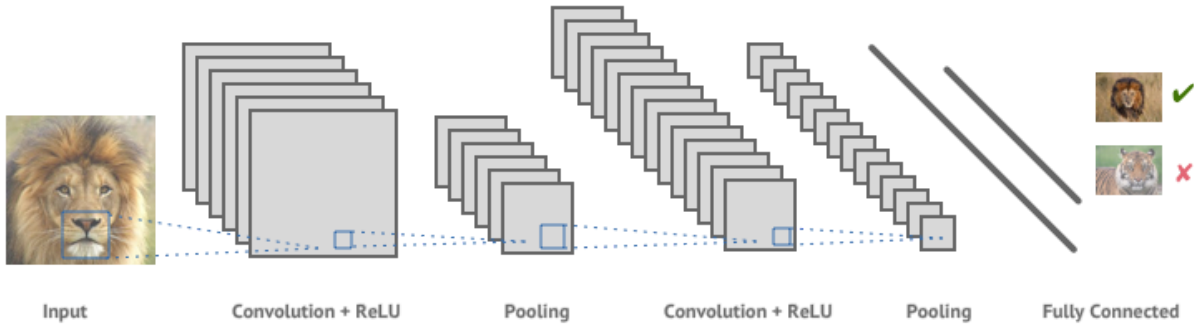


Figura 14 - Exemplo de arquitetura de uma rede neural convolucional[5].

Por fim, na camada de saída, os neurônios são conectados de forma global e não mais local, para gerar o vetor de saída *O(x)*. Com isso é calculada uma função de custo, que pode ser, por exemplo, o erro médio quadrático $C = \frac{1}{N}\Sigma(|Y(X) - O(X)|^2)$, onde *Y(x)* é o valor esperado associado à entrada. A ideia é diminuir este custo o máximo possível através de iterações de treinamento.

As duas primeiras etapas deste processo de três etapas são a propagação direta (*forward propagation*) e o cálculo de custo, explicados acima. A terceira etapa é a propagação reversa (*backpropagation*). De acordo com o cálculo do custo, os valores dos pesos são atualizados para frente em cada camada de acordo com a equação $w_1 = w_0 - \eta \frac{\delta C}{\delta w}$, onde η é a taxa de aprendizado, e $\frac{\delta C}{\delta w}$ é a taxa de variação do custo. Se for escolhida apropriadamente a taxa de aprendizado, cada iteração fará com que a função de custo diminua, chegando ao mínimo após algumas dezenas de iterações de treinamento (chamadas de *epochs*).

Este algoritmo iterativo é chamado de *Gradient Descent*, literalmente descida do vetor gradiente, buscando o mínimo da função de custo, exemplificado na Figura 15.

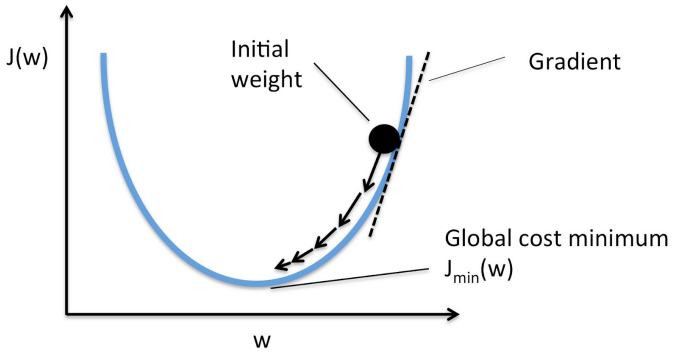


Figura 15 - Representação visual do algoritmo iterativo *Gradient Descent*[6].

[5] Figura extraída de "Machines that can see: Convolutional Neural Networks" https://shafeentejani.github.io/2016-12-20/convolutional-neural-nets/. Acessado em 31 out. 2020.
[6] Figura extraída de “Gradient Descent and Stochastic Gradient Descent” http://rasbt.github.io/mlxtend/user_guide/general_concepts/gradient-optimization/. Acessado em 03 nov. 2020.

Não é necessário implementar do zero todo este procedimento. Esta etapa do projeto também foi feita em Python, que conta com uma plataforma de aprendizado de máquina chamada Keras em TensorFlow, de onde podem ser importadas bibliotecas com funções e métodos necessários para a continuidade do projeto. Keras é a implementação em alto nível de redes neurais sobre TensorFlow, permitindo a criação destas redes em poucas linhas.

Antes de começar o treinamento, é preciso deixar as imagens armazenadas de forma conveniente para a rede neural usá-las no treinamento. Até então, elas estavam salvas em 24 arquivos **.pkl**, representando um `DataFrame` de 2136 linhas e 4725 colunas. Para estruturar um treinamento via *pipeline*, onde não é necessário carregar as imagens na memória em grande quantidade, foi preciso salvar as imagens dos *Kanjis* numa pasta local, separadas em 2136 pastas diferentes representando as classes a serem reconhecidas, com 4725 imagens do *Kanji* respectivo.

Um outro aspecto ainda não mencionado da fase de treinamento é a prática de separar uma quantidade de imagens para o treinamento e outra (menor) para validação, simulando a verificação da acurácia da rede neural em dados não vistos, evitando o *overfitting*.

Então, primeiramente, foram criadas duas pastas locais chamadas ‘train’ e ‘valid’, onde 80% das imagens seriam alocadas à primeira e o resto à segunda. Dentro de cada pasta foram criadas as pastas 0001 a 2136 representando a classe a ser reconhecida do *Kanji*, preenchidas com zero para manter a ordem alfabética padrão. Feito isso, foi feito um código em Python para ler os arquivos **.pkl** e salvar as imagens dos *Kanjis* no formato **.jpg**.

```
total_columns = 4725
train_perc = 0.8
train_columns = random.sample(range(1,total_columns+1), round(total_columns*train_perc))

def traverse_rows(row, folder):
    kanji = str(row.name).zfill(4)
    output_path = 'C:\\Users\\ferna\\Desktop\\tg\\%s\\%s\\{}.jpg' % (folder, kanji)
    for i in row.index:
        cv2.imwrite(output_path.format(i), row[i])

source_folder = 'C:\\Users\\ferna\\Desktop\\tg\\source'
source_list = os.listdir(source_folder)
paths_list = [source_folder+'\\'+file for file in source_list]
for path in paths_list:
    print(path)
    df = pd.read_pickle(path)
    df.loc[:, df.columns.isin(train_columns)].apply(lambda x: traverse_rows(x, 'train'), axis=1)
    df.loc[:, df.columns.isin(train_columns)==False].apply(lambda x: traverse_rows(x, 'valid'), axis=1)
    del df
```

Figura 16 - Código para salvar as **JPGs** dos *Kanjis* no HD da máquina.

Segundo o código da Figura 16, a primeira parte é definir a quantidade total de imagens por *Kanji* (colunas do `DataFrame`) e o percentual de quanto deve ser alocado à pasta de treinamento para o resto servir para validação. O variável `train_columns` é um *array* contendo 80% dos números das colunas de 1 a 4725

de forma aleatória, para que o treinamento não deixe de lado as distorções aplicadas nos últimos 20%, coletando dados de forma generalizada.

A função `traverse_rows` salva todas as imagens contidas na linha do `DataFrame` no formato **.jpg**. Como é preciso salvar cada linha do `DataFrame` em pastas diferentes, foi usado o método `apply` e não mais o `applymap`, para que a função possa percorrer linha por linha e salvá-las adequadamente. Para decidir onde salvar entre a pasta 'train' e a 'valid', bastou apenas filtrar o `DataFrame` de acordo com a variável `train_columns`. Uma indicação de como as imagens foram divididas nas duas pastas pode ser vista nos números das imagens da Figura 17.

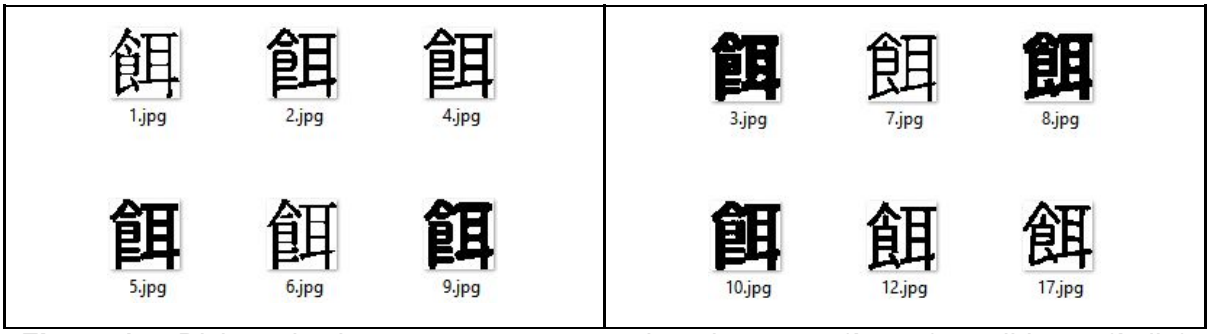


Figura 17 - Divisão das imagens entre a pasta de treinamento (à esq.) e validação (à dir.) para o *Kanji* 餌 (*esa*, isca).

Com estes diretórios preparados, pôde-se finalmente partir para o início do treinamento da rede neural. A aplicação prática da CNN com uso de diretórios locais foi baseada nos exercícios básicos de classificação de imagem com *Machine Learning* do Google Developers[7].

[7] Disponível em https://developers.google.com/machine-learning/practica/image-classification/exercise-1. Acessado em 31 out. 2020.

```
In [1]:
from tensorflow.keras.preprocessing.image import ImageDataGenerator

train_datagen = ImageDataGenerator(rescale=1./255)
valid_datagen = ImageDataGenerator(rescale=1./255)

train_dir = 'C:\\Users\\ferna\\Desktop\\tg\\train'
train_generator = train_datagen.flow_from_directory(
        train_dir,
        target_size=(48, 48),
        color_mode="grayscale",
        batch_size=20,
        shuffle=True,
        class_mode='categorical')

valid_dir = 'C:\\Users\\ferna\\Desktop\\tg\\valid'
valid_generator = valid_datagen.flow_from_directory(
        valid_dir,
        target_size=(48, 48),
        color_mode="grayscale",
        batch_size=20,
        shuffle=True,
        class_mode='categorical')
```

```
Found 8074080 images belonging to 2136 classes.
Found 2018520 images belonging to 2136 classes.
```

Figura 18 - Preparação para o treinamento da CNN com Keras.

No código da Figura 18, inicia-se fazendo o *setup* do pré-processamento das imagens com a classe `ImageDataGenerator`, reescalando os valores para 0 e 1 em vez de 0 e 255, para que os valores se mantenham comportados (tenham variação de magnitude menores) durante as atribuições dos pesos. Criando um objeto para cada pasta, foi então usado o método `flow_from_directory`, que recebe os parâmetros para o local da pasta, dimensões da imagem, modo de cor, tamanho de *batch*, embaralhamento e modo de classificação.

Modo de cor define-se entre colorido ou em escala de cinza, este último o selecionado já que não há opção binária. Tamanho de *batch* define o lote de quantas amostras precisam ser usadas para atualizar os parâmetros da rede uma única vez. Essa divisão em *batches* é feita para poder fazer mais *updates* nos parâmetros em vez de esperar passar pelo número total de amostras antes de uma atualização, algo que favorece um algoritmo iterativo de convergência, além de requerer menos memória.

Definir o *shuffle* (embaralhamento) como verdadeiro ajuda os *batches* a terem estatísticas gerais mais próximas as estatísticas do total de amostras (não ocorrer polarização na amostragem, que vicia parâmetros a uma região específica), favorecendo a convergência. E por último, o modo de classificação é definido como categórico, já que existem mais de 2 classes no problema.

Outra função ainda não mencionada do método `flow_from_directory` é sua capacidade de gerar ainda mais *data augmentation* durante a própria etapa de treinamento no *background*, aumentando ainda mais a variação dos dados.

```
from tensorflow.keras import layers
from tensorflow.keras import Model

img_input = layers.Input(shape=(48, 48, 1))

x = layers.Conv2D(64, 5, activation='tanh')(img_input)
x = layers.MaxPooling2D(2)(x)
x = layers.Conv2D(128, 5, activation='tanh')(x)
x = layers.MaxPooling2D(2)(x)
x = layers.Conv2D(256, 5, activation='tanh')(x)
x = layers.MaxPooling2D(2)(x)
x = layers.Flatten()(x)
x = layers.Dense(1024, activation='tanh')(x)
output = layers.Dense(2136, activation='softmax')(x)

model = Model(img_input, output)
model.summary()
```

```
Layer (type)                 Output Shape              Param #
=================================================================
input_9 (InputLayer)         [(None, 48, 48, 1)]       0
_________________________________________________________________
conv2d_26 (Conv2D)           (None, 44, 44, 64)        1664
_________________________________________________________________
max_pooling2d_24 (MaxPooling (None, 22, 22, 64)        0
_________________________________________________________________
conv2d_27 (Conv2D)           (None, 18, 18, 128)       204928
_________________________________________________________________
max_pooling2d_25 (MaxPooling (None, 9, 9, 128)         0
_________________________________________________________________
conv2d_28 (Conv2D)           (None, 5, 5, 256)         819456
_________________________________________________________________
max_pooling2d_26 (MaxPooling (None, 2, 2, 256)         0
_________________________________________________________________
flatten_6 (Flatten)          (None, 1024)              0
_________________________________________________________________
dense_12 (Dense)             (None, 1024)              1049600
_________________________________________________________________
dense_13 (Dense)             (None, 2136)              2189400
=================================================================
Total params: 4,265,048
Trainable params: 4,265,048
Non-trainable params: 0
```

Figura 19 - Definição das camadas da CNN e resumo de seus parâmetros.

Na Figura 19, a estruturação da rede neural começa pela camada de entrada, onde é preciso passar as dimensões das imagens amostradas, além de uma terceira dimensão (representando o canal preto e branco) permitindo a multiplicação correta em cada nó do grafo.

A seguir, são definidas as camadas profundas, sempre uma em função da anterior. Neste caso, as camadas de convolução 2D e ativação são feitas em um mesmo passo. Por exemplo, a primeira camada de convolução foi composta por 64 neurônios, uma janela de tamanho 5 para a convolução, e função de ativação por tangente hiperbólica (tanh).

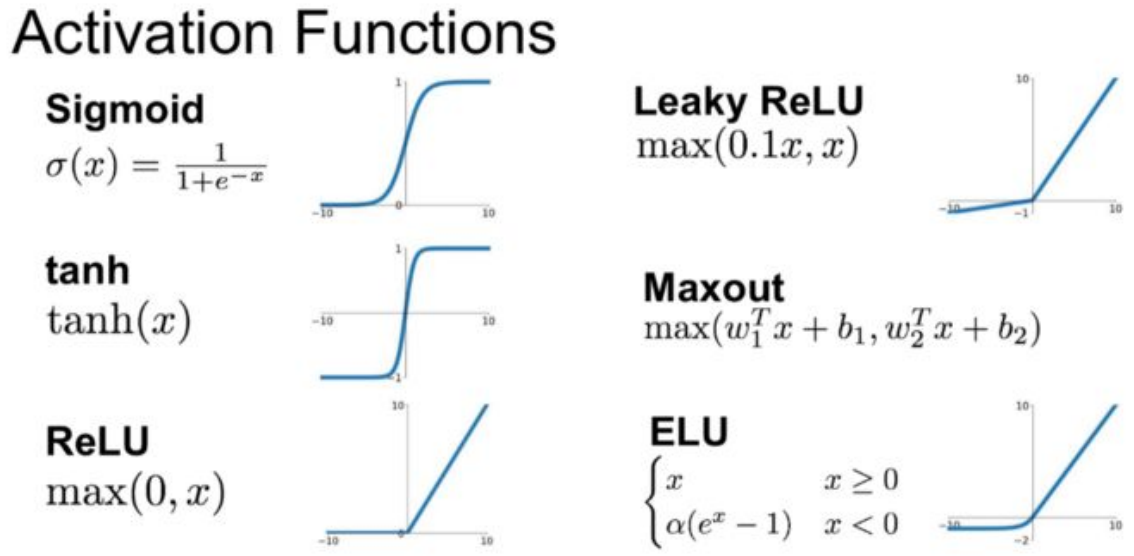


Figura 20 - Possíveis funções de ativação[8] pós etapa de convolução.

Antes de ter-se optado pela ativação em tanh e por essa estrutura da rede em geral, foram feitos vários testes empíricos com quantidades de camadas e de nós diferentes, incluindo funções de ativação diferentes, onde algumas delas estão à mostra na Figura 20. A ReLU, mencionada anteriormente como exemplo, também foi testada, mas foi descartada por causar anulamento de muitos neurônios da rede e travar a convergência do treinamento em um percentual muito baixo.

A ativação em tanh se mostrou empiricamente satisfatória. É possível discutir que, comparada com a ReLU, a tanh possui curvatura antissimétrica, que se relaciona mais favoravelmente a um problema de reconhecimento de imagem, já que classificação de imagem depende de uma relação espacial entre pixels em níveis acima do que uma equação do primeiro grau (linear) é capaz de representar.

A quantidade de parâmetros adicionados após uma convolução, indicados na Figura 19, pode ser calculada pela equação $N_{param} = n_{out}(n_{in}.(h.w)_{kernel} + 1)$. O primeiro termo da soma representa as ponderações de todos os pesos na entrada de um nó, e o segundo o *bias*. A primeira convolução é feita com um kernel de 5x5, o $(h.w)_{kernel}$. O número de canais de entrada da primeira etapa n_{in} é 1, já que o processo ainda está na imagem de entrada com um único canal. Se n_{out} é o número de nós de saída 64, tem-se $64(1 \times 5 \times 5 + 1) = 1664$ parâmetros.

A etapa de *max pooling* é apenas para generalizar as características da imagem e diminuir custo computacional, e não adiciona parâmetros. A segunda convolução, agora com n_{in} de valor 64 já que é o número de nós conectados na entrada, tem-se mais $128(64 \times 5 \times 5 + 1) = 204928$ parâmetros, e assim por diante.

No fim da rede, é preciso converter o formato da saída em um vetor linha, usando a função `Flatten`. Antes da separação final nos 2136 neurônios de saída,

[8] Figura extraída de "Introduction to Different Activation Functions for Deep Learning" https://medium.com/@shrutijadon10104776/survey-on-activation-functions-for-deep-learning-9689331ba092. Acessado em 02 nov. 2020.

é feita uma penúltima camada plana com 1024 nós, para se obter uma ponderação mais complexa antes da classificação final. Aqui o cálculo dos parâmetros leva em consideração formato de saída da etapa anterior, 2x2 em 256 nós, havendo a multiplicação por 1024 e também soma por 1024 seguindo o padrão de peso e *bias*, resultando em $2 \times 2 \times 256 \times 1024 + 1024 = 1049600$ parâmetros. A última camada de classificação adiciona $1024 \times 2136 + 2136 = 2189400$ parâmetros.

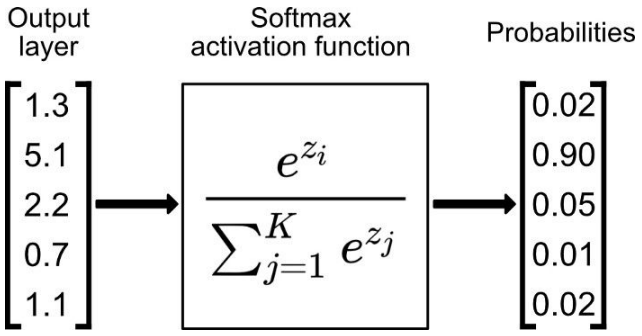


Figura 21 - Função de ativação softmax[9] para a última camada.

Nota-se que a função de ativação da última camada definida na Figura 19 contendo os 2136 nós representando as classes dos *Kanjis* não possui função de ativação tanh, e sim softmax, elucidada na Figura 21. Foi mencionado anteriormente que o vetor de saída da rede neural deve ser um array contendo um número para cada probabilidade de classificação para aquele *Kanji*, e é justamente o que a função softmax faz, normalizando a soma dos valores em 1, simulando uma ponderação probabilística.

Outro fator importante é o fato de que o número de nós aumenta conforme a posição da camada. Isso é feito para que os primeiros parâmetros sirvam para ponderar e dividir características mais gerais da imagem, e ir aumentando a complexidade conforme a profundidade, requerendo mais parâmetros.

```
from tensorflow.keras.optimizers import RMSprop
from tensorflow.keras.optimizers import Adam

model.compile(loss='categorical_crossentropy',
              optimizer=Adam(lr=0.0001),
              metrics=['acc'])
```

Figura 22 - Compilação do modelo com a função de perda e o otimizador.

Na compilação do modelo da Figura 22, a função de perda (ou custo) selecionada foi a *Categorical Crossentropy Loss*, de equação $C = -\Sigma (y.log\ p)$ (*y* é o valor da saída esperada, *p* é a probabilidade resultante), que é a função padrão utilizada para problemas de classificação com mais de duas classe e com vetores de saída padronizados em *one-hot encoding*.

[9] Figura extraída de "Softmax Activation Function Explained" https://towardsdatascience.com/softmax-activation-function-explained-a7e1bc3ad60. Acessado em 02 nov. 2020.

O otimizador é justamente a função que tentará achar o mínimo da função de custo. Entre as opções possíveis estão *Stochastic Gradient Descent*, versão estocástica do algoritmo explicado na fundamentação teórica, onde os pesos são atualizados a cada amostra e não após a passagem de todas as amostras. É geralmente mais rápido que outros otimizadores, mas tende a ser menos preciso principalmente em problemas com grande número de parâmetros.

Foram testados então os dois otimizadores importados na figura, o RMSprop (*Root Mean Square Propagation*) e o Adam (*Adaptative Moment Estimation*). RMSprop foi testado com várias configurações de redes neurais diferentes, e não mostrou resultados promissores, travando a acurácia do reconhecedor perto dos 50%. Ao trocar o otimizador para o Adam, o problema foi resolvido.

De acordo com a documentação do Keras, ambos incluem um vetor extra chamado de 'momento' que faz ajustes finos no vetor gradiente da função de otimização de custo, porém só o Adam possui momentos tanto de primeira como de segunda ordem, enquanto o RMSprop possui apenas um momento padrão. Assume-se que este pode ser um dos motivos para a diferença de performance neste caso.

Por fim, a métrica escolhida para avaliar o desempenho do classificador foi o 'ACC' (*Accuracy*), que dá uma estimativa de quantas predições foram corretas na etapa de treinamento. Não é usada no algoritmo iterativo de treinamento.

```
history = model.fit_generator(
      train_generator,
      steps_per_epoch=100,
      epochs=100,
      validation_data=valid_generator,
      validation_steps=100,
      verbose=2)
```

Figura 23 - Método `fit_generator` para iniciar o treinamento.

O método `fit_generator` da Figura 23 inicia o treinamento, recebendo os dois geradores de treino e validação criados anteriormente. Decidiu-se por 100 *epochs* de treinamento, onde cada *epoch* fará 100 iterações de treinamento (`steps_per_epoch`) definidas anteriormente com o tamanho de *batch* 20. Número de passos de validação foi definido como o mesmo número de *epochs* para se ter informação em todas elas, e `verbose` define que informações são impressas no console do Python durante o treino.

```
Epoch 1/100
100/100 - 148s - loss: 7.6752 - acc: 0.0025 - val_loss: 7.6636 - val_acc: 0.0000e+00
Epoch 2/100
100/100 - 124s - loss: 7.6488 - acc: 0.0015 - val_loss: 7.6270 - val_acc: 5.0000e-04
Epoch 3/100
100/100 - 169s - loss: 7.5712 - acc: 0.0035 - val_loss: 7.5137 - val_acc: 0.0015
Epoch 4/100
100/100 - 148s - loss: 7.4634 - acc: 0.0030 - val_loss: 7.3971 - val_acc: 0.0055
Epoch 5/100
100/100 - 157s - loss: 7.2987 - acc: 0.0060 - val_loss: 7.2105 - val_acc: 0.0080

Epoch 96/100
100/100 - 106s - loss: 0.8708 - acc: 0.8415 - val_loss: 0.8670 - val_acc: 0.8455
Epoch 97/100
100/100 - 135s - loss: 0.8801 - acc: 0.8485 - val_loss: 0.8449 - val_acc: 0.8465
Epoch 98/100
100/100 - 105s - loss: 0.7998 - acc: 0.8550 - val_loss: 0.8205 - val_acc: 0.8520
Epoch 99/100
100/100 - 124s - loss: 0.8237 - acc: 0.8545 - val_loss: 0.8235 - val_acc: 0.8555
Epoch 100/100
100/100 - 144s - loss: 0.8293 - acc: 0.8620 - val_loss: 0.8243 - val_acc: 0.8520
```

Figura 24 - Informações impressas no console das 5 primeiras e 5 últimas *epochs* do treino.

O código executou por cerca de 4 horas. No fim das 100 *epochs*, indicado na Figura 24, o reconhecedor obteve aproximadamente 85% de acurácia. Os valores gerais de perda e acurácia por *epoch* ficaram salvas na variável `history`, possibilitando plotagem de gráficos mostrando o comportamento durante o treino.

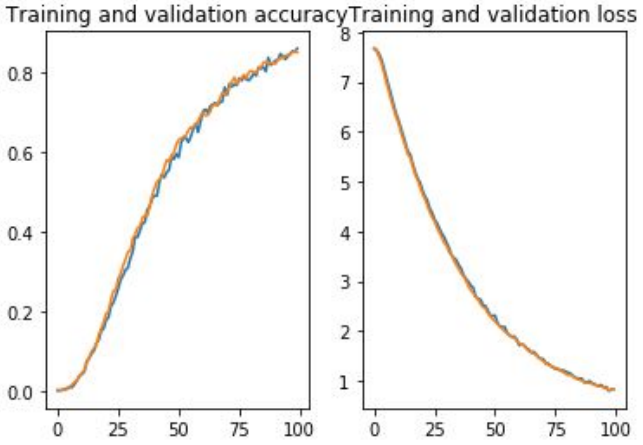


Figura 25 - Curvas para acurácia e perda do reconhecedor (treino em azul, validação em laranja).

Cada gráfico da Figura 25 possui 2 curvas cada, uma para os dados usados no treino e outros da parte de validação. Como são basicamente coincidentes, é concluído que não houve *overfitting*, tendo a mesma acurácia em amostras não vistas. Outro detalhe importante no gráfico é que as curvas ainda não atingiram seu limite onde começam a ficar planas, havendo margem para mais treinamento e aumento de acurácia do reconhecedor.

Portanto, com o mesmo objeto model anterior, foi executado novamente o método `fit_generator`, dessa vez definindo `epochs` = 120 e um parâmetro extra `initial_epoch` = 101, para que o treino ocorresse por mais 20 iterações.

```
Epoch 116/120
100/100 - 104s - loss: 0.6739 - acc: 0.8785 - val_loss: 0.6392 - val_acc: 0.8820
Epoch 117/120
100/100 - 130s - loss: 0.6432 - acc: 0.8760 - val_loss: 0.6411 - val_acc: 0.8830
Epoch 118/120
100/100 - 106s - loss: 0.6327 - acc: 0.8775 - val_loss: 0.6298 - val_acc: 0.8805
Epoch 119/120
100/100 - 140s - loss: 0.6412 - acc: 0.8770 - val_loss: 0.6192 - val_acc: 0.8880
Epoch 120/120
100/100 - 147s - loss: 0.6006 - acc: 0.8950 - val_loss: 0.6262 - val_acc: 0.8840
```

Figura 26 - Últimas iterações da continuação do treino.

Com estas iterações extras indicadas na Figura 26, foi possível aumentar a acurácia do classificador para cerca de 90%, que em um problema com 2136 classes onde há a possibilidade de também verificar a segunda, terceira e assim por diante probabilidades de classificação da imagem, julgou-se satisfatório.

Com todos os parâmetros armazenados no objeto `model`, foi possível salvar um arquivo kanji_classifier**.h5** (formato usado no armazenamento de grandes quantidades de dados numéricos hierárquicos) com o método `save`. O arquivo do grafo final ficou com tamanho próximo a 50 MB, pondo fim a etapa de treinamento.

4 - DESENVOLVIMENTO DO APLICATIVO DE CELULAR COM BASE NO RECONHECEDOR

Smartphones possuem poder de processamento consideravelmente menor do que *notebooks* e *desktops*. Um formato mais conveniente para este tipo de dispositivo em vez do **.h5** é o **.tflite**, e a própria biblioteca do TensorFlow em Python já tem esta conversão implementada, que pode ser feita em poucas linhas, como na Figura 27. O novo arquivo gerado tem tamanho 17 MB, comparado com aproximadamente 50 MB em formato **.h5**.

```
import tensorflow as tf

converter = tf.lite.TFLiteConverter.from_keras_model_file(
    'C:\\Users\\ferna\\Desktop\\tg\\kanji_classifier.h5')
tflite_model = converter.convert()
open('C:\\Users\\ferna\\Desktop\\tg\\kanji_classifier.tflite',
    "wb").write(tflite_model)
```

Figura 27 - Passos para a conversão de **.h5** para **.tflite**.

O *framework* de desenvolvimento de aplicativos escolhido foi o Flutter, que é um *kit* de desenvolvimento de aplicativos de código aberto criado pela Google em 2017, e sua linguagem utilizada é o Dart, que também é mantida pela mesma empresa. Entre suas maiores vantagens sobre outros meios de desenvolvimento são sua capacidade de desenvolver para Android e iOS usando um único código, e os chamados *hot reload* e *hot restart*, que são maneiras rápidas de se recarregar ou reiniciar um *preview* de uma interface em desenvolvimento, algo que só existia em desenvolvimento web e era um gargalo importante no desenvolvimento *mobile*.

Uma maneira de se criar um ambiente de desenvolvimento Flutter no desktop é usando o software Visual Studio Code da Microsoft. Ele é um editor de código leve com ferramentas que permitem a inclusão do *framework* em seu meio. Para instalar o Flutter nele, basta pesquisar a opção 'Extensions: Install Extensions' no Command Palette e procurar a opção Flutter na lista de extensões. Essa opção também instala o *plugin* para a linguagem Dart.

Também no Command Palette do Visual Studio Code, pesquisando e clicando na opção "Flutter: Run Flutter Doctor", é possível ver a situação dos passos necessários para fazer o *setup* do desenvolvimento no *framework*.

```
[√] Flutter (Channel stable, v1.17.5, on Microsoft Windows [versÃ£o 10.0.18362.836], locale ja-JP)
    • Flutter version 1.17.5 at C:\Users\ferna\Documents\Flutter SDK
    • Framework revision 8af6b2f038 (4 months ago), 2020-06-30 12:53:55 -0700
    • Engine revision ee76268252
    • Dart version 2.8.4

[√] Android toolchain - develop for Android devices (Android SDK version 28.0.3)
    • Android SDK at C:\Users\ferna\AppData\Local\Android\sdk
    • Platform android-28, build-tools 28.0.3
    • Java binary at: C:\Program Files\Android\Android Studio\jre\bin\java
    • Java version OpenJDK Runtime Environment (build 1.8.0_202-release-1483-b03)
    • All Android licenses accepted.

[√] Android Studio (version 3.5)
    • Android Studio at C:\Program Files\Android\Android Studio
    • Flutter plugin version 43.0.1
    • Dart plugin version 191.8593
    • Java version OpenJDK Runtime Environment (build 1.8.0_202-release-1483-b03)

[√] VS Code (version 1.50.0)
    • VS Code at C:\Users\ferna\AppData\Local\Programs\Microsoft VS Code
    • Flutter extension version 3.10.2

[!] Connected device
    ! No devices available
```

Figura 28 - *Output* do console do VS Code quando executado o comando Flutter Doctor.

Da lista das 5 ferramentas necessárias da Figura 28, o Flutter em si acabou de ser instalado via Command Palette do VS Code. 'Android toolchain' se refere ao SDK (*software development kit*) do Android que é instalado junto ao Android Studio. O desenvolvimento no VS Code usa este SDK, e não foi usado o Android Studio em si, mesmo sua instalação sendo necessária para obtenção do SDK.

Por fim, é checado se há algum dispositivo móvel conectado para se fazer os testes enquanto se desenvolve. Aqui é possível escolher entre criar um dispositivo virtual carregado na memória do PC ou configurar um *smartphone* real para executar estes testes. A vantagem de se usar dispositivos virtuais é a possibilidade de testar várias versões diferentes do Android, além de haver variações nas dimensões da tela. Porém, foi decidido usar um smartphone real para poupar custo computacional, alocando a memória necessária dos testes para o próprio aparelho celular. O dispositivo usado foi o Moto E5, na versão 8.0.0 do Android, como na Figura 29.

```
[√] Connected device (1 available)
    • moto e5 • 0047596150 • android-arm • Android 8.0.0 (API 26)
```

Figura 29 - Resultado positivo para verificação de dispositivos conectados.

Para liberar a permissão de uso para desenvolvimento do celular, o passo-a-passo envolve clicar 7 vezes na opção 'Sobre o telefone' nas configurações, que libera a opção 'Programador' no menu do sistema, permitindo ativar este modo, como na Figura 30.

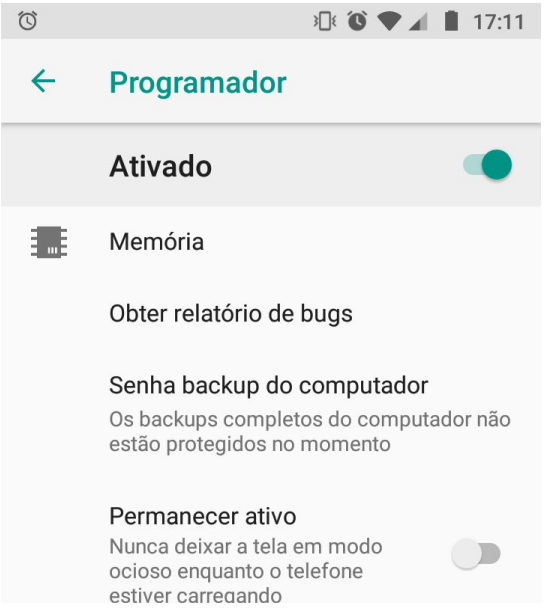


Figura 30 - Opção 'Programador' ativada.

Assim, é possível criar o projeto em Flutter e começar a desenvolver. A etapa de extrair a imagem do Canvas para usar no reconhecedor em formato **.tflite**, explicado melhor mais adiante, foi baseada no artigo (dividido em 5 partes) '*Handwriting number recognizer with Flutter and TensorFlow*'[10] de Sergio Fraile.

[10] Disponível em https://medium.com/flutter-community/handwriting-number-recognizer-with-flutter-and-tensorflow-part-i-414157b7574f. Acessado em 11 nov. 2020.

A ideia, porém, é fazer uma interface diferente da estruturada nestes artigos, e com funcionalidades adicionais. O aplicativo proposto nos artigos possui apenas uma página, mostrando somente a tela a desenhar o dígito e a previsão de maior probabilidade.

Para o caso do projeto, haverá duas telas. A principal, onde é feito o desenho do *Kanji* a ser reconhecido, e onde é mostrado vários *Kanji* possíveis em posição decrescente de probabilidade. Tocando no *Kanji* desejado, passa-se para a segunda tela, onde é mostrada informações de pronúncia e significado. A diferença no resultado final pode ser vista na Figura 31.

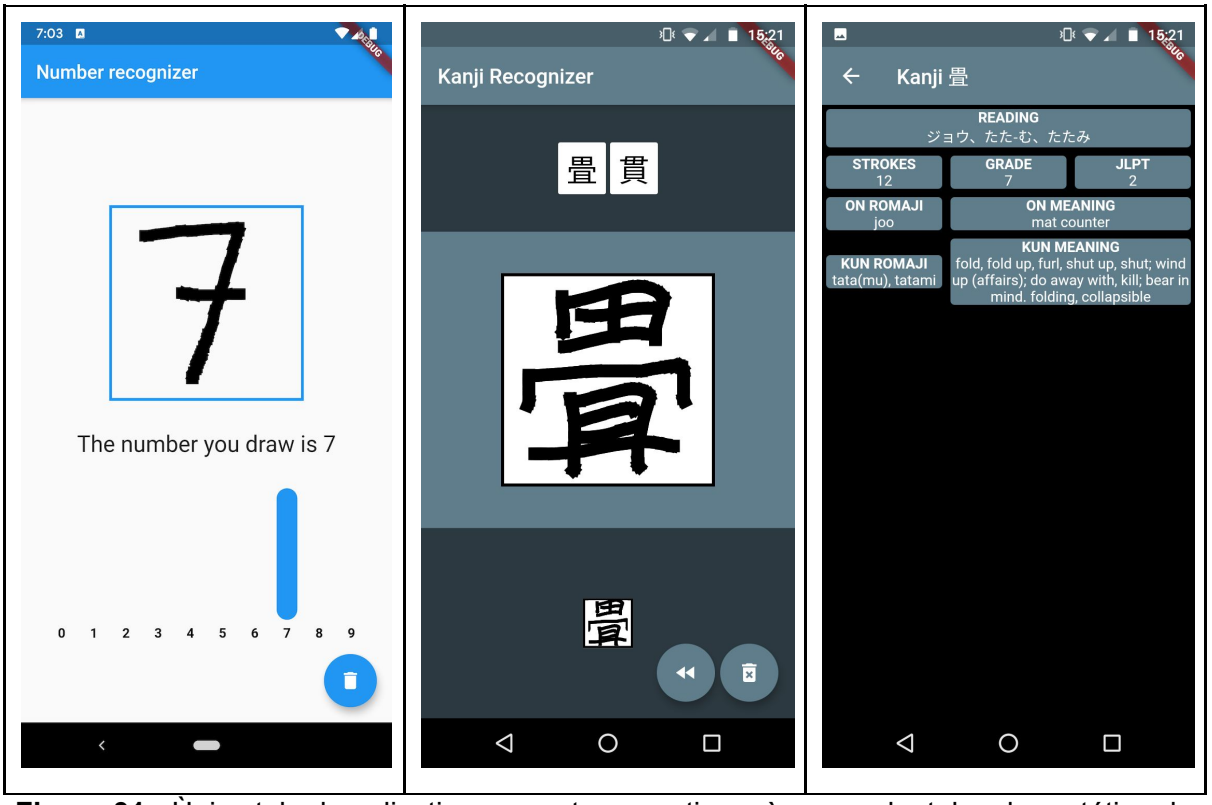


Figura 31 - Ùnica tela do aplicativo proposto nos artigos, à esquerda; telas do protótipo do aplicativo desenvolvido no projeto, no centro e à direita.

Para criar um projeto Flutter no Visual Studio Code, é preciso abrir o Command Palette, e procurar e selecionar a opção ‘Flutter: New Project’, sendo necessário apenas escolher um nome para ele. No caso, o nome escolhido foi ‘kanji_recognizer’. Selecionado o nome, é criado automaticamente um aplicativo *template* com todos os arquivos necessários para a continuidade do desenvolvimento, listados na Figura 32.

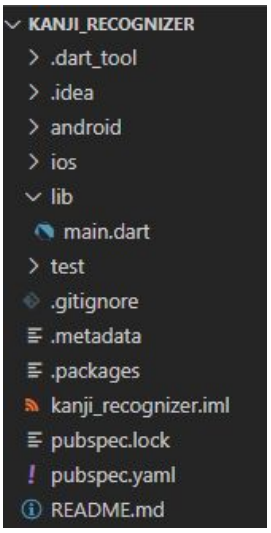


Figura 32 - Lista de pastas e arquivos criados automaticamente junto a um novo projeto Flutter.

Os dois arquivos mais importantes dessa lista, e praticamente os únicos a serem modificados, são os arquivos lib/main**.dart** e pubspec**.yaml**. Com o smartphone plugado, ativada a opção Programador e permitida a transferência de arquivos por USB, ao se apertar a tecla F5 no VS Code é possível executar o aplicativo *template* através do arquivo main**.dart**, sem ter feito nenhuma modificação no código.

Figura 33 - Tela do aplicativo *template*.

Figura 34 - Parte 1 do código contido no arquivo main**.dart**.

```
29  class _MyHomePageState extends State<MyHomePage> {
30    int _counter = 0;
31    void _incrementCounter() {
32      setState(() {
33        _counter++;
34      });
35    }
36
37    @override
38    Widget build(BuildContext context) {
39      return Scaffold(
40        appBar: AppBar(
41          title: Text(widget.title),
42        ), // AppBar
43        body: Center(
44          child: Column(
45            mainAxisAlignment: MainAxisAlignment.center,
46            children: <Widget>[
47              Text('You have pushed the button this many times:'),
48              Text('$_counter', style: Theme.of(context).textTheme.headline4),
49            ], // <Widget>[]
50          ), // Column
51        ), // Center
52        floatingActionButton: FloatingActionButton(
53          onPressed: _incrementCounter,
54          tooltip: 'Increment',
55          child: Icon(Icons.add),
56        ), // FloatingActionButton
57      ); // Scaffold
58    }
59  }
```

Figura 35 - Parte 2 do código contido no arquivo main**.dart**.

O arquivo executa o que está na função main (linha 3 da Figura 34). Para executar o aplicativo, usa-se a função runApp, que recebe um *widget*. *Widget* é a peça base para todo aplicativo em Flutter. Eles são criados uns dentro dos outros e têm função tanto de posição (`Column`, `Center`, etc) como de interface visível (`Button`, `Text`, etc).

Eles também podem ser divididos em duas categorias relacionadas a estado: *stateless* e *stateful*. *Stateless* é uma designação para *widgets* que não alteram seus estados, enquanto *stateful* refere-se a *widgets* que atualizam seus estados conforme programado ou por alguma interação do usuário.

No caso do *app template*, o *widget* que começa a cadeia é *stateless*. É o que está definido na classe da linha 7 da Figura 34, que é do tipo `MaterialApp`. Ele define um padrão inicial do aplicativo, incluindo o título que é mostrado quando o aplicativo é visto na lista de *apps* abertos do *smartphone*, e seu tema que define a configuração padrão de cores.

O parâmetro `home` (linha 16, Figura 34) recebe o que deve ser o conteúdo da tela, que no caso do *template* é a barra superior contendo o título visível do aplicativo aberto, o texto indicando quantas vezes o botão foi pressionado, e o próprio botão, como pode ser visto na Figura 33. Como o estado precisa ser alterado para atualizar a contagem, o widget deve ser *stateful*, como foi criado em `MyHomePage` na linha 21 da Figura 34.

Para o tipo `StatefulWidget`, sempre é criado em conjunto um outro *widget* do tipo `State` que herda a própria classe, como foi criado na linha 26 da Figura 34 com o método `createState` e definido na linha 29 da Figura 35. É nessa classe State que é definido o que se quer que o `StatefulWidget` faça na prática.

Ela começa definindo um método `_incrementCounter`, cuja função é atualizar a variável `_counter` conforme a quantidade de vezes que o botão for pressionado. Cada método que atualiza o estado do *widget* precisa chamar outro método chamado `setState`, que avisa ao Flutter que sua renderização precisa ser atualizada.

Feito isso, já é possível definir na prática do que deve ser composta a interface deste *widget*, através do *override* do método `build`, como foi feito também no começo para o *widget stateless*. O que foi definido para ser retornado (linha 39, Figura 35) é um `Scaffold`, que é justamente um padrão de tela de um aplicativo com uma barra superior para o título (`appBar`), o conteúdo no restante da tela (`body`) e um botão caso seja necessária alguma ação do usuário (`floatingActionButton`).

O conteúdo foi definido iniciando a cadeia com o *widget* `Center` que centraliza seu conteúdo de acordo com a regra definida em `mainAxisAlignment`. No caso, foi definido como `mainAxisAlignment.center` fazendo com que os dois objetos textos filhos fiquem colados no centro, mas outras opções poderiam ser utilizadas para posicioná-los com espaçamento vertical simétrico entre os dois textos, por exemplo.

Seus *widgets* filhos, como mencionado, são dois `Text`, um para a frase que não se altera, e outro para mostrar o valor da variável `_counter`, indicando quantas vezes foi apertado o botão. Por fim, é definido o botão flutuante no canto direito inferior, que recebe uma função que será executada quando pressionado (nesse caso, aumentar a contagem), além do ícone que se quer que o botão tenha.

Tudo isso foi explicado para dar uma ideia de como Flutter funciona, mas este arquivo main**.dart** criado automaticamente foi completamente reescrito no desenvolvimento do aplicativo reconhecedor de *Kanji*. Além disso, foram criados vários outros arquivos **.dart** na pasta ‘lib’ ao lado do main**.dart** para ter o desenvolvimento do código de forma mais modularizada.

Figura 36 - Nova pasta ‘assets’.

Como visto na Figura 36, foi criada a pasta ‘assets’ contendo os três arquivos externos necessários do projeto. O arquivo ‘kanji_recognizer**.tflite**’ é o reconhecedor que havia sido convertido para o formato mais leve anteriormente.

O arquivo ‘kanji_table**.json**’ é uma tabela no formato *JavaScript Object Notation*, que é uma maneira eficiente de se armazenar dados hierárquicos para serem usados como consulta em várias linguagens de programação. Para a criação desta tabela, mais uma vez foi usada a linguagem Python, desta vez para montar a tabela com as informações dos *Kanjis* que apareceriam na segunda tela do aplicativo.

Consultando novamente o site *Kanji Database*, foi possível baixar uma tabela em formato **CSV** sendo selecionadas informações como pronúncia e significado, manuseada em Python e salva em formato **JSON**, para poder ser usada no aplicativo em Flutter.

	KANJI	STROKES	GRADE	JLPT	READING	ON_ROMAJI	ON_TRANSLATION	KUN_ROMAJI	KUN_TRANSLATION
0	亜	7	7	1	ア	a	rank next, come after, Asia, sub-, -ous (in ac...	-	-
1	哀	9	7	1	アイ、あわ-れ、あわ-れむ	ai	pity, have mercy on, sympathize with	awa(re), awa(remu)	pity, have mercy on, sympathize with, grief, s...
2	挨	10	7	1	アイ	ai	push open	-	-
3	愛	13	4	2	アイ	ai	love, affection, favorite	-	-
4	曖	17	7	-	アイ	ai	dark; not clear	-	-

Figura 37 - Tabela customizada para uso como consulta no aplicativo.

Seguindo a mesma ordem de indexação dos *Kanjis* feita no treinamento, as colunas da tabela da Figura 37 contêm informações incluindo o número de traços (`STROKES`), em qual ano escolar japonês é ensinado o *Kanji* (`GRADE`) e em qual teste de proficiência da língua japonesa o *Kanji* é cobrado (`JLPT`, *Japanese Language Proficiency Test*).

O *Kanji* pode ter duas categorias de leitura/pronúncia: a de origem chinesa chamada *onyomi* e a de origem japonesa chamada *kunyomi*. Portanto, é importante a tabela conter informações das duas categorias de leitura juntas aos significados, indicados nas colunas `ON_ROMAJI`, `ON_TRANSLATION`, `KUN_ROMAJI` e `KUN_TRANSLATION`. Também foi adicionada a coluna `READING` contendo as leituras em Katakana para *onyomi* e em Hiragana para *kunyomi*, como é padronizado no ensino.

O último arquivo da pasta ‘assets’ é um arquivo **.txt** contendo as *labels* da previsão dos 2136 *Kanjis*. O resultado da previsão retorna um índice de 0 a 2135 junto à sua probabilidade, e para saber a qual símbolo este índice se refere é necessário manter esta lista de *Kanjis* em um **.txt** com a mesma ordem, com um símbolo por linha.

Para permitir que o aplicativo utilize estes três arquivos externos, é preciso listá-los no último bloco do arquivo pubspec**.yaml**, encabeçado por ‘flutter’ e ‘assets’, seguindo a tabulação e formatação como indicado na Figura 38.

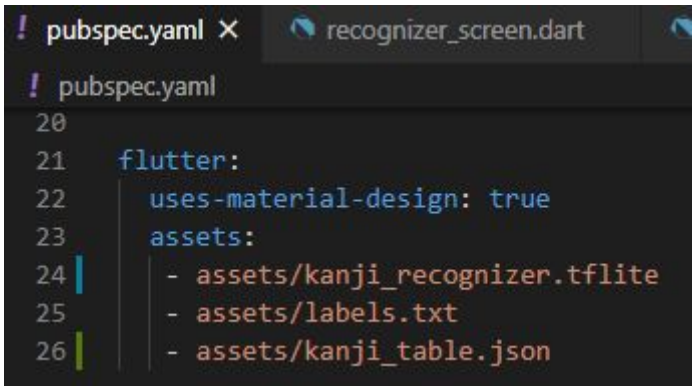


Figura 38 - Importando arquivos externos para o aplicativo.

Para poder utilizar bibliotecas externas e baixar suas dependências, também é usado o arquivo pubspec**.yaml**. As bibliotecas aprovadas para uso em Dart e em Flutter estão listadas no site pub.dev[11]. Para incluí-las no projeto, basta listar seus nomes definidos pelo site no bloco de código encabeçado por 'dependencies' no arquivo pubspec**.yaml**, seguidas de suas respectivas versões desejadas, como na Figura 39. Salvando as alterações, o Visual Studio Code executa um comando automático para baixar as dependências necessárias relativas às bibliotecas.

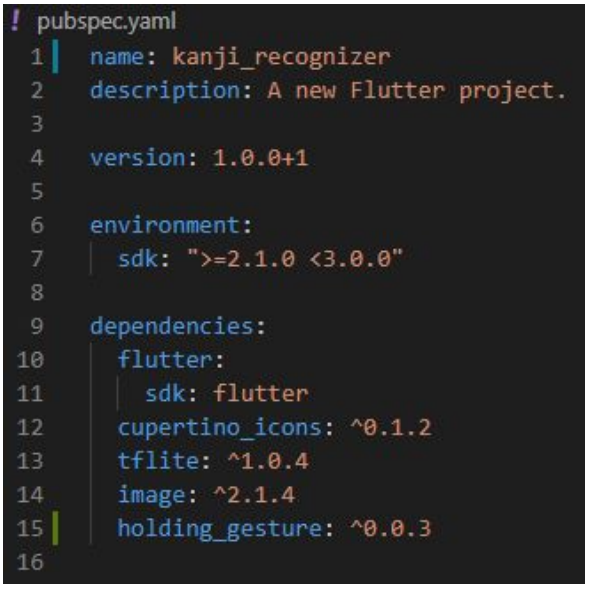


Figura 39 - Listando as bibliotecas tflite, image e holding_gesture em 'dependencies' (flutter e cupertino_icons já vêm por padrão).

As três bibliotecas importadas foram: 'tflite', que permite o uso de arquivos **.tflite** para fazer previsões incluindo o reconhecimento de imagem; 'image', que é necessário para manusear variáveis de imagem de uma maneira que o 'tflite' reconheça; e 'holding_gesture', que foi usado para alocar a tarefa de "rebobinar" um tracejado do usuário ao se manter um botão pressionado.

[11] Disponível em https://pub.dev/. Acesso em 16 nov. 2020.

Na página do ‘tflite’ do site pub.dev, é mencionada a necessidade de se fazer uma alteração no arquivo ‘android/app/build**.gradle**’ para o funcionamento correto do pacote (Figura 40), além de ser necessário adicionar um arquivo labels**.txt**, como já feito. Para os outros dois pacotes não foram necessárias outras alterações.

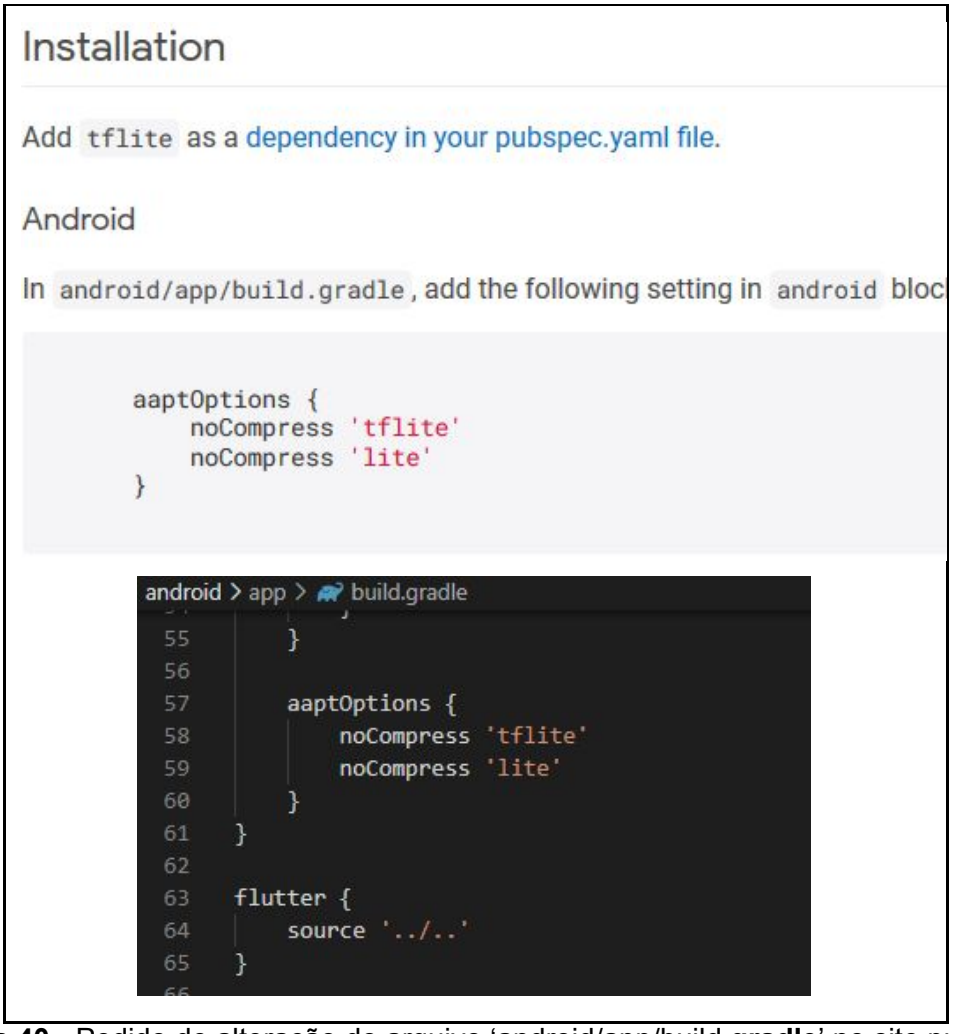


Figura 40 - Pedido de alteração do arquivo ‘android/app/build**.gradle**’ no site pub.dev (acima), e arquivo alterado (abaixo).

Terminadas todas essas preparações, partiu-se para o desenvolvimento propriamente dito do projeto, representado pelo código nos arquivos da pasta ‘lib’, indicado na Figura 41.

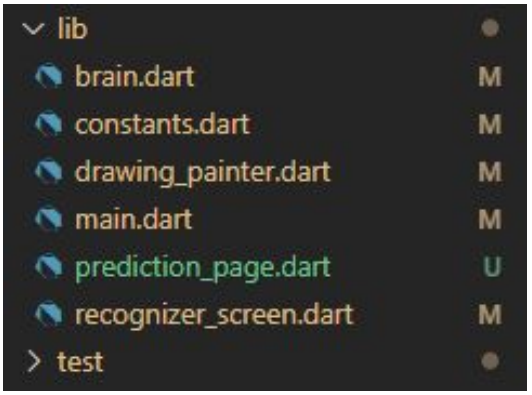


Figura 41 - Arquivos de código na pasta ‘lib’ do projeto finalizado.

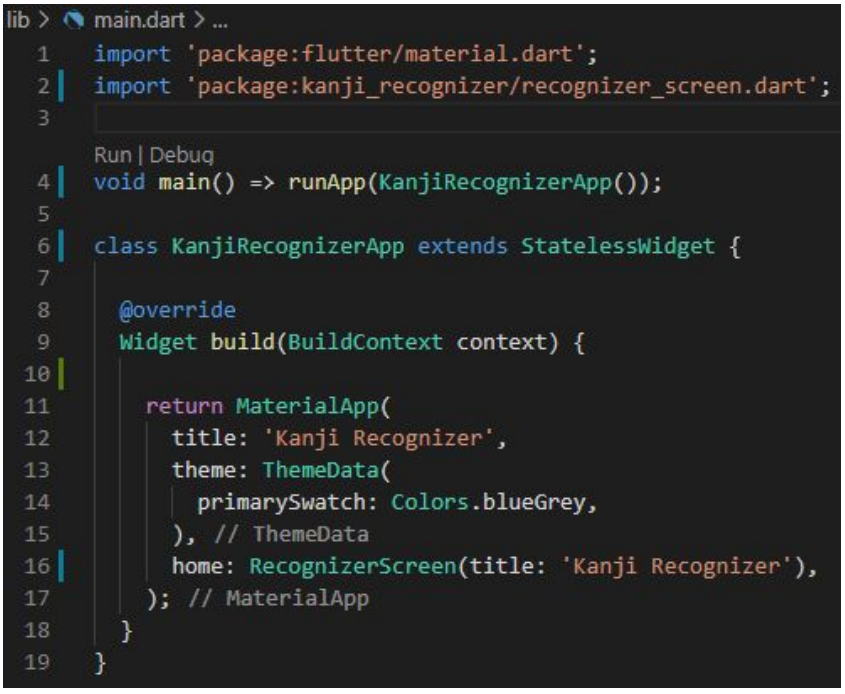


Figura 42 - Código de todo o arquivo main**.dart**.

Assim como no aplicativo *template*, a execução do *app* começa na função main do arquivo main**.dart**, que executa a função `runApp`. O parâmetro recebido é o *widget* sem estado `KanjiRecognizerApp`, que é o *widget* que engloba todos os outros. O resto do código apenas define que o app é do tipo `MaterialApp`, cuja toda a interface está definida no *widget* com estado `RecognizerScreen`, definido no arquivo recognizer_screen**.dart**, importado no cabeçalho da Figura 42.

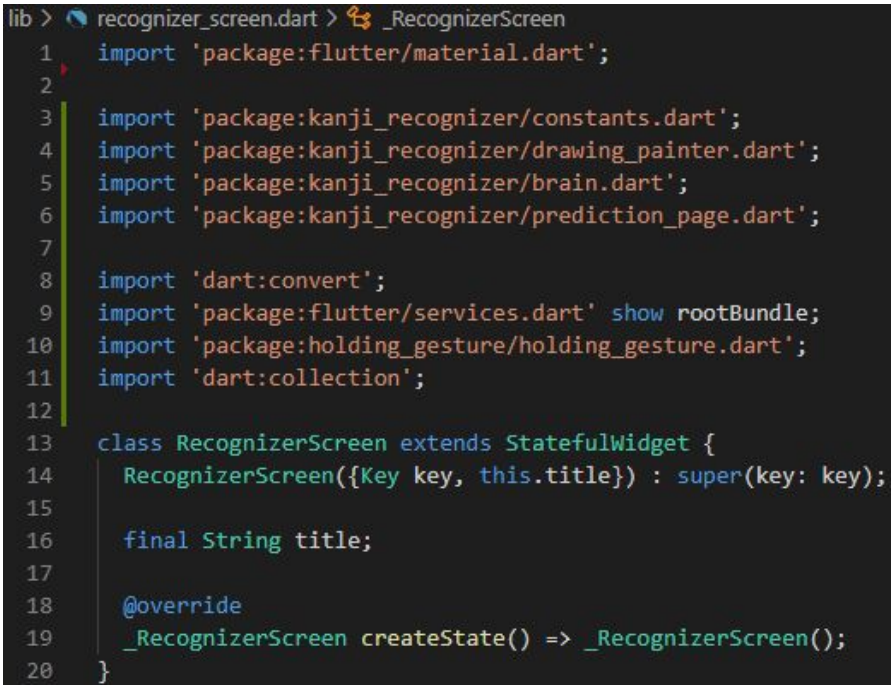


Figura 43 - Definição do widget `RecognizerScreen`, com as importações necessárias e a criação do estado `_RecognizerScreen`.

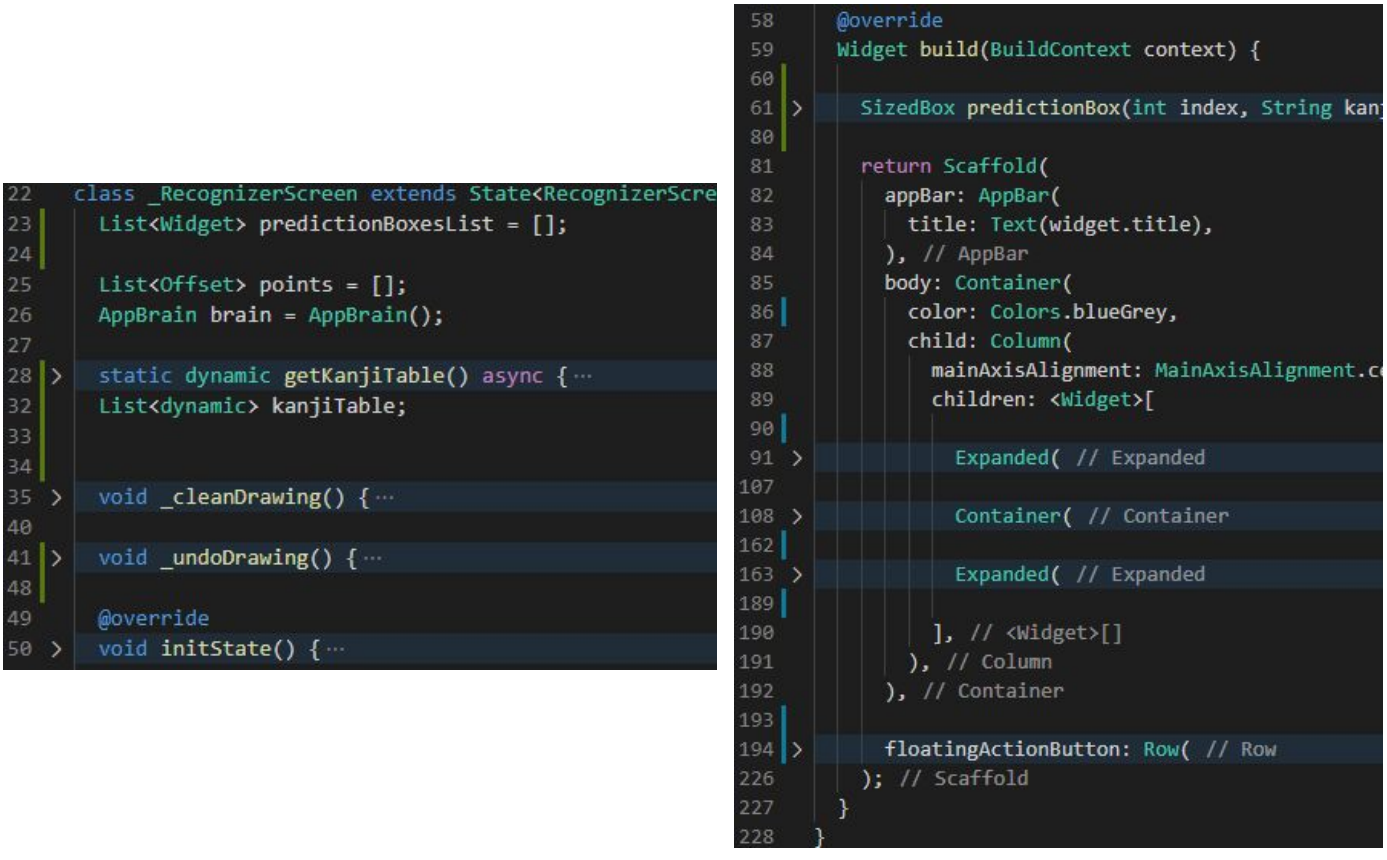


Figura 44 - Todo o bloco de código para o estado `_RecognizerScreen` com funções e *widgets* internos ocultados.

O código em recognizer_screen**.dart** das Figuras 43 e 44 é composto pela definição do *widget* com estado `RecognizerScreen` e seu respectivo estado `_RecognizerScreen`. O foco primeiramente é nos três widgets que compõem o corpo do `Scaffold`: o primeiro `Expanded`, o `Container` e o segundo `Expanded` (Figura 44 à direita), cujo *layout* resultante pode ser visto na Figura 45.

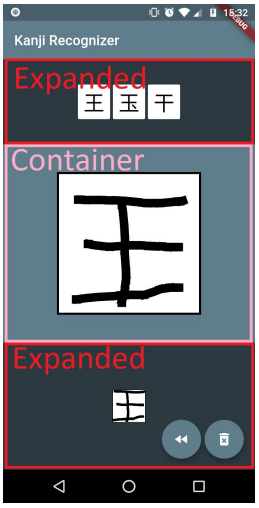
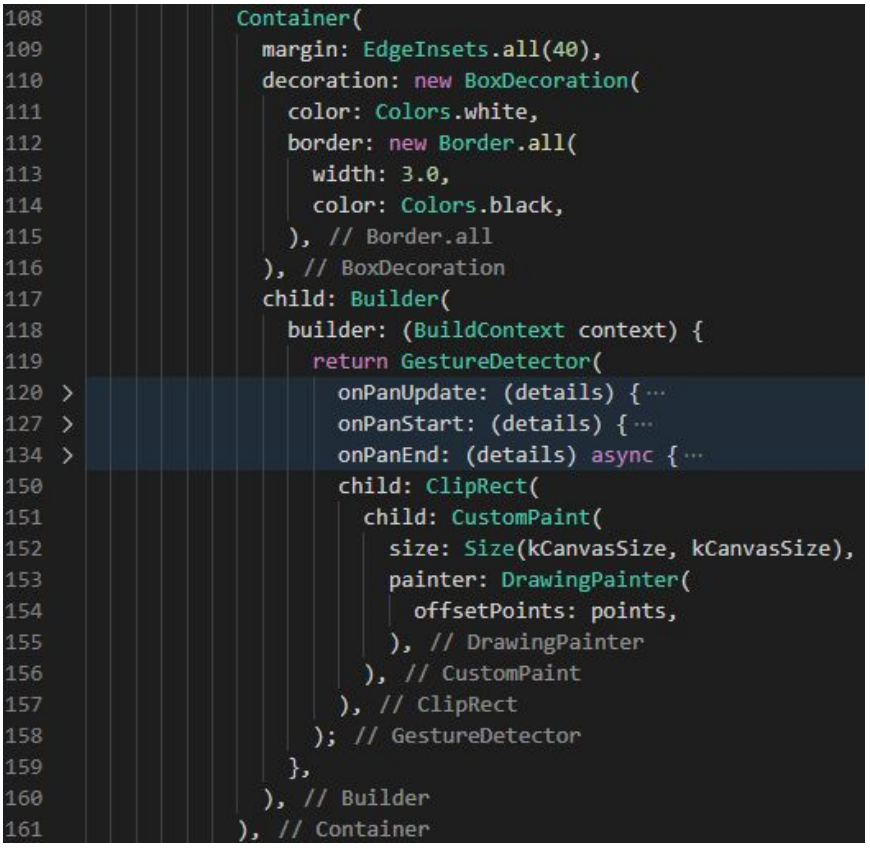


Figura 45 - Interface da tela principal destacando os três *widgets* principais.

O primeiro `Expanded` contém a lista de *Kanjis* previstos pelo reconhecedor em ordem decrescente de probabilidade (da esquerda para a direita). O `Container` possui o canvas onde o usuário desenha o *Kanji* que se quer reconhecer. O segundo `Expanded` contém a imagem desenhada já feito o *crop* do espaço em branco envoltório e redimensionada para 48x48, para indicar ao usuário que não é preciso usar o canvas inteiro na hora de desenhar o símbolo.



```
108          Container(
109            margin: EdgeInsets.all(40),
110            decoration: new BoxDecoration(
111              color: Colors.white,
112              border: new Border.all(
113                width: 3.0,
114                color: Colors.black,
115              ), // Border.all
116            ), // BoxDecoration
117            child: Builder(
118              builder: (BuildContext context) {
119                return GestureDetector(
120 >                onPanUpdate: (details) {···
127 >                onPanStart: (details) {···
134 >                onPanEnd: (details) async {···
150                  child: ClipRect(
151                    child: CustomPaint(
152                      size: Size(kCanvasSize, kCanvasSize),
153                      painter: DrawingPainter(
154                        offsetPoints: points,
155                      ), // DrawingPainter
156                    ), // CustomPaint
157                  ), // ClipRect
158                ); // GestureDetector
159              },
160            ), // Builder
161          ), // Container
```

Figura 46 - Estruturação dos widgets dentro do `Container`.

O fluxo começa pelo desenho do símbolo pelo usuário no canvas localizado no `Container` (linha 108, Figura 46). Define-se uma margem de 40 pixels nas quatro direções. No parâmetro `decoration` é definido fundo branco e borda preta, que são aplicados no canvas (`CustomPaint` da linha 151), englobado por um `ClipRect` para que caso o usuário saia da área delimitada não sejam gerados traços externos. O tamanho do canvas é `kCanvasSize` = 200 pixels, definido em constants**.dart** (Figura 47).

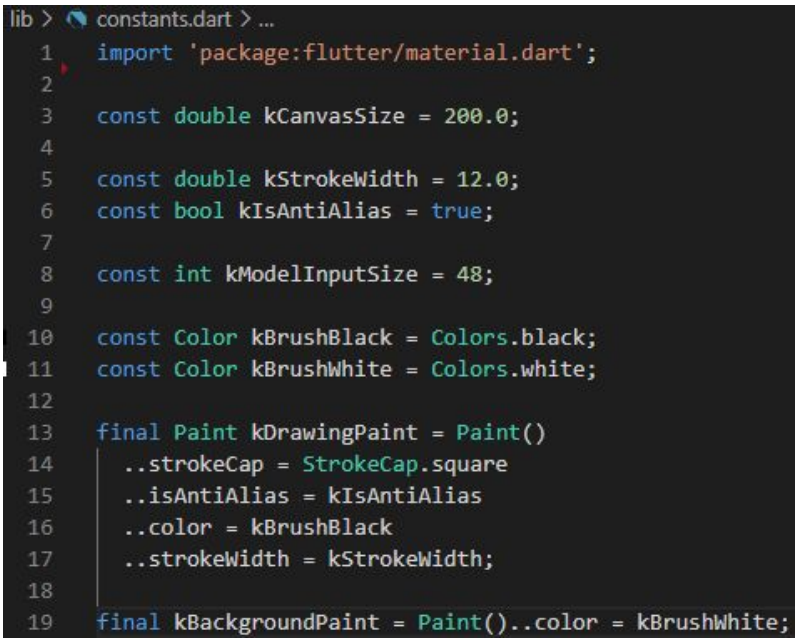


Figura 47 - Todas as constantes definidas em constants**.dart**.

O `CustomPaint` do `Container` precisa receber um `painter`, onde é definida a regra de como é feito o desenho no canvas. `DrawingPainter` é uma implementação do *widget* `CustomPainter` (*widget* diferente do `CustomPaint`) feita no arquivo drawing_painter**.dart**, Figura 48. Ela recebe um *array* de pontos no formato `Offset(x, y)` e o desenha de acordo com a regra implementada. A lista de pontos é a variável `points`, gerada no `GestureDetector`, explicado mais adiante.

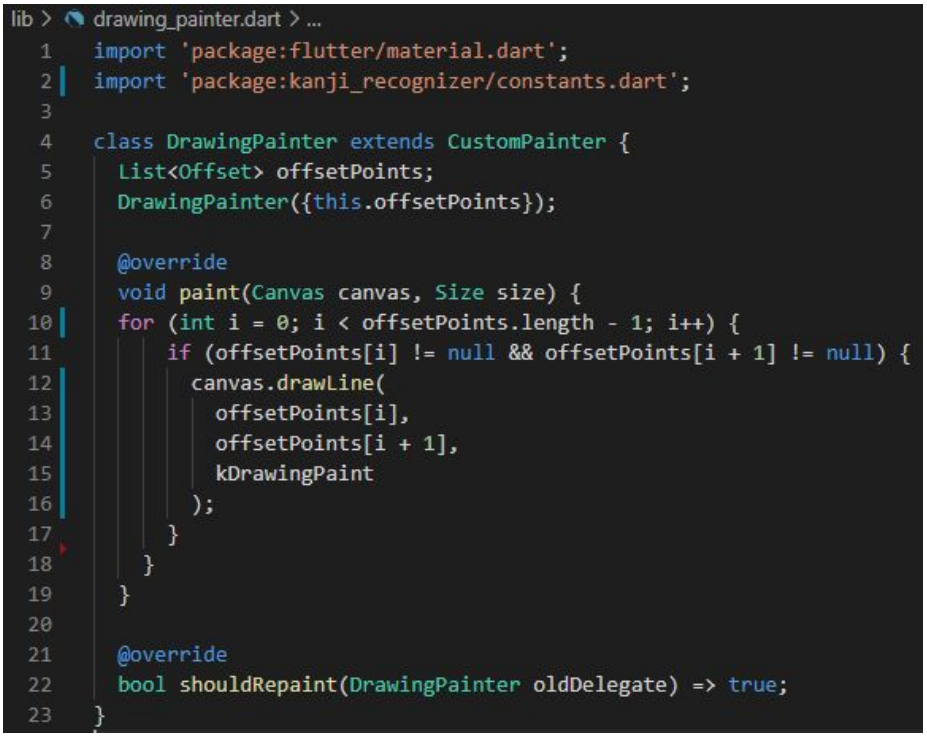


Figura 48 - Código para o arquivo drawing_painter**.dart**.

Foi definido que para instanciar a classe `DrawingPainter`, é preciso passar uma lista de pontos (*array* de `Offsets`) nas linhas 5 e 6 da Figura 48, similar ao que foi feito para definir o título do aplicativo em `RecognizerScreen`, por exemplo.

A documentação da classe `CustomPainter` diz que é necessária a implementação de dois métodos: `paint` e `shouldRepaint`. Em `paint`, é definida a regra de como gerar a pintura na tela, que é usar o método `drawLine` do canvas em todos os pontos adjacentes da lista. O método precisa receber 2 pontos e uma "tinta" (classe `Paint`), definida no arquivo das constantes (linha 13, Figura 47) com pincel quadrado, anti-alias entre os pontos (redução de serrilhamento), cor preta e grossura do pincel 12 pixels.

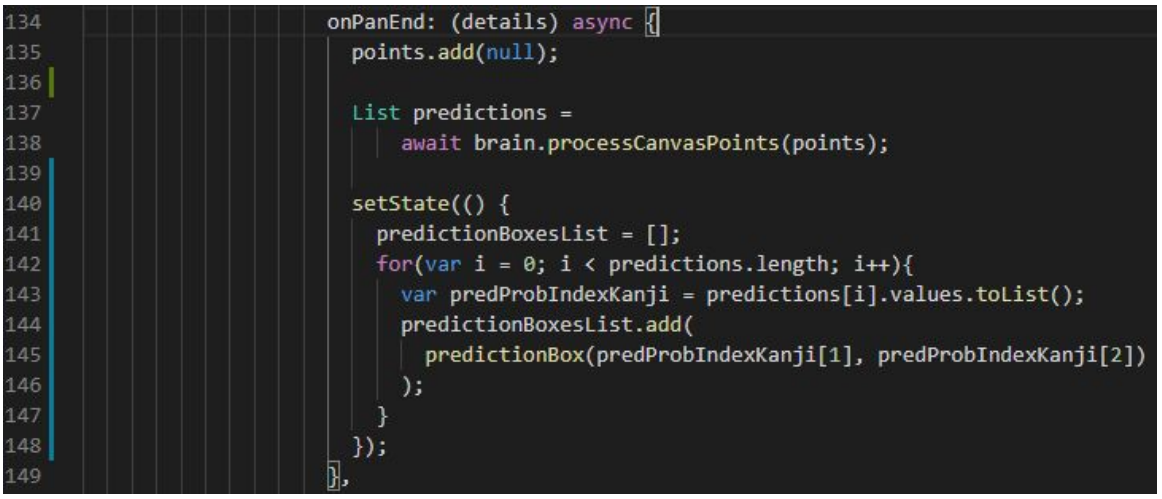
Para o método `shouldRepaint`, que define quando o desenho deve ser atualizado na tela, neste caso basta só definir como sempre verdadeiro, para que o fluxo do reconhecimento de imagem seja feito o tempo todo.

```
119                 return GestureDetector(
120                   onPanStart: (details) {
121                     setState(() {
122                       RenderBox renderBox = context.findRenderObject();
123                       points.add(
124                           renderBox.globalToLocal(details.globalPosition));
125                     });
126                   },
127                   onPanUpdate: (details) {
128                     setState(() {
129                       RenderBox renderBox = context.findRenderObject();
130                       points.add(
131                           renderBox.globalToLocal(details.globalPosition));
132                     });
133                   },
134 >                 onPanEnd: (details) async {···
150                   child: ClipRect(
```

Figura 49 - Parâmetros do `GestureDetector` dentro do `Container`.

É no bloco do `GestureDetector` (Figura 49) que são coletados os pontos para preenchimento da lista de `Offsets(x, y)` da variável `points`. O evento `onPanStart` é quando o usuário dá o primeiro toque na tela; `onPanUpdate` é quando a posição do dedo muda enquanto ainda em contato com a tela; e `onPanEnd` é quando o dedo deixa de estar em contato.

Nos dois primeiros eventos, são coletados os pontos de acordo com a posição local, que podem variar tanto em **x** e **y** entre 0 e 200, de acordo com as dimensões definidas no `CustomPaint`, *widget* filho.



```
134                 onPanEnd: (details) async {
135                   points.add(null);
136
137                   List predictions =
138                       await brain.processCanvasPoints(points);
139
140                   setState(() {
141                     predictionBoxesList = [];
142                     for(var i = 0; i < predictions.length; i++){
143                       var predProbIndexKanji = predictions[i].values.toList();
144                       predictionBoxesList.add(
145                         predictionBox(predProbIndexKanji[1], predProbIndexKanji[2])
146                       );
147                     }
148                   });
149                 },
```

Figura 50 - Tarefas alocadas ao evento `onPanEnd`.

No evento `onPanEnd` (Figura 50), ou seja, quando o usuário termina de desenhar um traço, primeiramente é adicionado um ponto nulo na lista de pontos. Isso é feito para que o fim de um traço e o começo do próximo traço não sejam interligados na hora do `DrawingPainter` fazer o desenho.

A seguir, é feita a predição do *Kanji* desenhado usando os pontos coletados do `GestureDetector` no fluxo de reconhecimento definido no objeto `brain`, instância da classe `AppBrain`, definida no arquivo brain**.dart**. É neste arquivo em que é usado o pacote ‘tflite’.

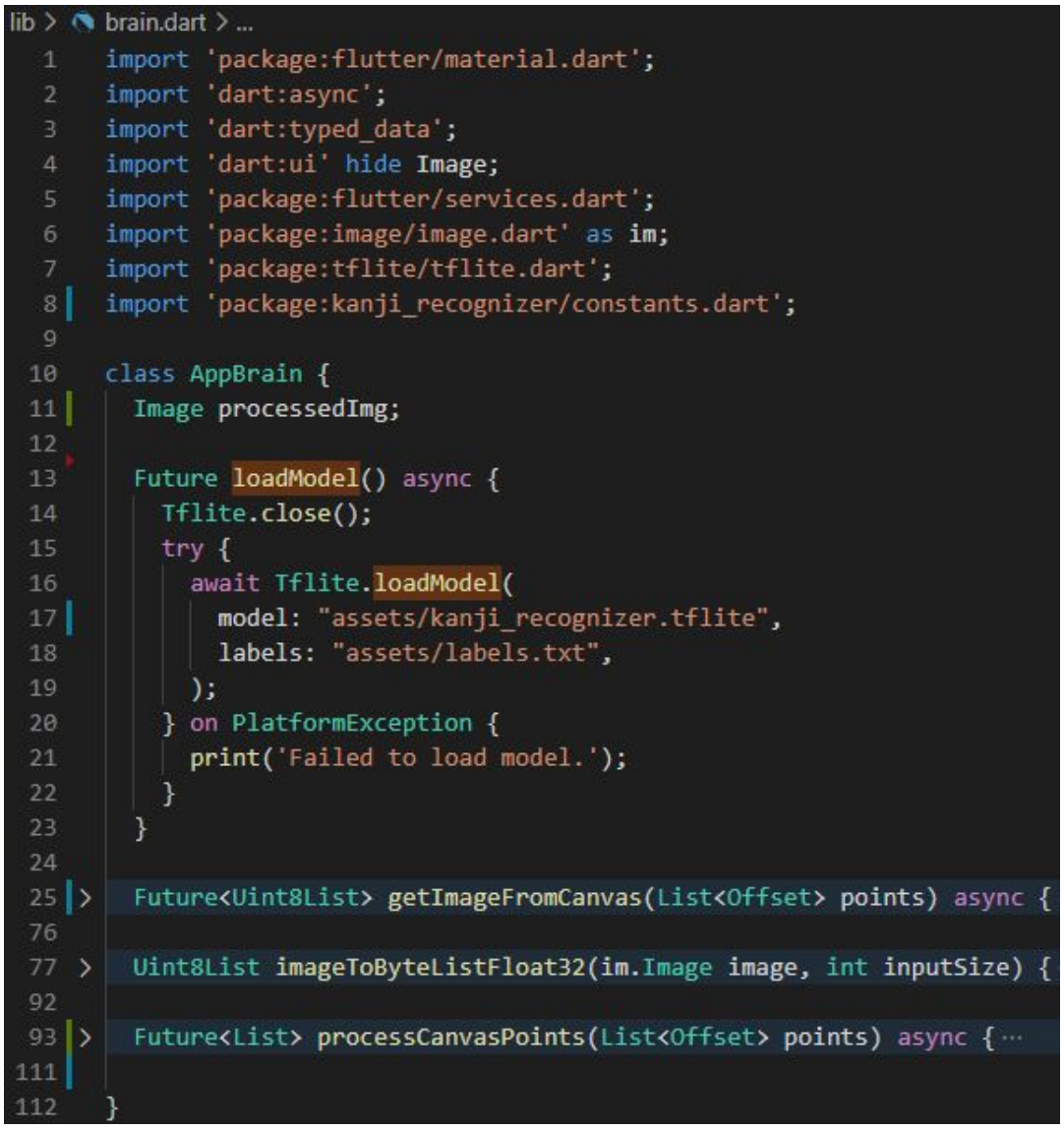


Figura 51 - Classe e métodos do arquivo brain**.dart**.

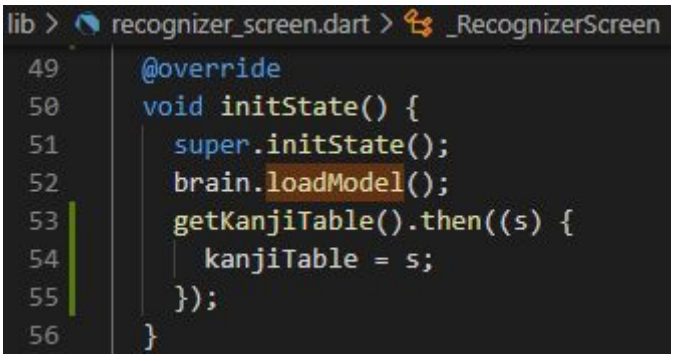


Figura 52 - Carregamento do modelo tflite no estado inicial em _RecognizerScreen**.dart**.

Para carregar o modelo, é preciso passar o local do grafo **.tflite** e da lista de **labels**, como feito nas linhas 17 e 18 da Figura 51. Isso é feito de forma assíncrona ao serem usadas as chaves *async* e *await*, para que o carregamento da interface do aplicativo possa ocorrer independentemente do fluxo de reconhecimento.

Na Figura 52, é feito o *override* do método `initState` para garantir que tanto o modelo como o **JSON** com a tabela para consulta estejam prontos antes de o usuário poder realizar alguma interação.

```
93    Future<List> processCanvasPoints(List<Offset> points) async {
94      Uint8List pngUint8List = await getImageFromCanvas(points);
95
96      processedImg = Image.memory(pngUint8List);
97
98      im.Image imImage = im.decodeImage(pngUint8List);
99      im.Image resizedImage = im.copyResize(
100       imImage,
101       width: kModelInputSize,
102       height: kModelInputSize,
103     );
104
105     return await Tflite.runModelOnBinary(
106       binary: imageToByteListFloat32(resizedImage, kModelInputSize),
107       numResults: 10,
108       threshold: 0.05,
109     );
110   }
```

Figura 53 - Método `processCanvasPoints` para gerar as previsões do reconhecimento da imagem.

```
DEBUG CONSOLE ···   Filter (e.g. text, !exclude)
I/flutter (14985): [{confidence: 0.2520943582057953, index: 2082, label: 涙},
{confidence: 0.0953340008854866, index: 1970, label: 友}, {confidence: 0.0873
030424118042, index: 832, label: 疾}]
```

Figura 54 - Print da variável `predictions`, resultado de `Tflite.runModelOnBinary`.

O método `processCanvasPoints` da Figura 53 recebe a lista de pontos e gera uma lista de pixels `uint8` (valores de 0 a 255) usando o método `getImageFromCanvas`, definido na mesma classe. Com essa lista de pixels, é gerada uma variável de imagem para ser usada como *preview* no `Expanded` inferior da tela principal, armazenada na variável `processedImg`.

Antes de executar o modelo reconhecedor, é gerada e redimensionada a imagem para 48x48, como definido na rede neural, usando o pacote externo ‘image’ importado previamente com `im.Image` nas linhas 98 e 99, com funções diferentes do `Image` padrão do Flutter.

Um dos métodos para reconhecimento de imagem mencionadas na documentação do pacote tflite é o `runModelOnBinary`, cuja saída pode ser vista no console da Figura 54. Nele, é preciso fazer a imagem passar pelo método `imageToByteListFloat32` para que a lista de pixels tenha tamanho fixo desejado de acordo com o formato de entrada do modelo, algo que não ocorre na

variável `pngUint8List`. Uma versão deste método pode ser encontrado na própria página do tflite no pub.dev, e a deste projeto pode ser vista na Figura 55.

```
77  Uint8List imageToByteListFloat32(im.Image image, int inputSize) {
78    var convertedBytes = Float32List(inputSize * inputSize);
79    var buffer = Float32List.view(convertedBytes.buffer);
80    int pixelIndex = 0;
81    for (var i = 0; i < inputSize; i++) {
82      for (var j = 0; j < inputSize; j++) {
83        var pixel = image.getPixel(j, i);
84        buffer[pixelIndex++] =
85            (im.getRed(pixel) + im.getGreen(pixel) + im.getBlue(pixel)) /
86                3 /
87                255.0;
88      }
89    }
90    return convertedBytes.buffer.asUint8List();
91  }
```

Figura 55 - Método `imageToByteListFloat32`.

Os outros dois parâmetros do método `runModelOnBinary` são o número máximo de resultados que se quer mostrar, e o limiar de probabilidade (linhas 107 e 108 da Figura 53). No caso, foi escolhido listar no máximo 10 resultados e com confiabilidade mínima de 5%. Como visto na Figura 53, o resultado do reconhecimento é uma lista contendo a probabilidade, o índice e a *label* de cada *Kanji*, seguindo as regras estipuladas em `numResults` e `threshold`.

```
25  Future<Uint8List> getImageFromCanvas(List<Offset> points) a
26    List drawnMin = [kCanvasSize, kCanvasSize];
27    List drawnMax = [0.0, 0.0];
28
29    for (var i=0; i<points.length; i++) {
30      if (points[i]!=null) {
31        if (points[i].dx < drawnMin[0]) drawnMin[0] = points[
32        if (points[i].dy < drawnMin[1]) drawnMin[1] = points[
33        if (points[i].dx > drawnMax[0]) drawnMax[0] = points[
34        if (points[i].dy > drawnMax[1]) drawnMax[1] = points[
35      }
36    }
37
38    List drawnSize = [drawnMax[0]-drawnMin[0]+kStrokeWidth,
39                      drawnMax[1]-drawnMin[1]+kStrokeWidth];
40    List<Offset> newPoints = points.toList();
41    for (var i=0; i<points.length; i++) {
42      if (points[i]!=null) {
43        newPoints[i] = points[i].translate(
44          -drawnMin[0]+kStrokeWidth/2, -drawnMin[1]+kStrokeWi
45      }
46    }
```

```
48    final recorder = PictureRecorder();
49    final canvas = Canvas(
50      recorder,
51      Rect.fromPoints(
52        Offset(0.0, 0.0),
53        Offset(drawnSize[0], drawnSize[1]),
54      ), // Rect.fromPoints
55    ); // Canvas
56
57    canvas.drawRect(
58      Rect.fromLTWH(0, 0, drawnSize[0], drawnSize[1]),
59      kBackgroundPaint,
60    );
61
62    for (int i = 0; i < newPoints.length - 1; i++) {
63      if (newPoints[i] != null && newPoints[i + 1] != null) {
64        canvas.drawLine(newPoints[i], newPoints[i + 1], kDrawin
65      }
66    }
67
68    final picture = recorder.endRecording();
69    final img = await picture.toImage(
70      drawnSize[0].toInt(), drawnSize[1].toInt());
71    final imgBytes = await img.toByteData(format: ImageByteForm
72    Uint8List pngUint8List = imgBytes.buffer.asUint8List();
73    return pngUint8List;
74  }
```

Figura 56 - Método `getImageFromCanvas`.

O método para gerar uma imagem a partir da lista de pontos é o `getImageFromCanvas`. No bloco à esquerda na Figura 56, são feitos cálculos para fazer o recorte do espaço em branco envoltório. O truque é encontrar os pontos mínimos e máximos desenhados nos eixos X e Y para fazer uma translação à origem usando os valores mínimos e definir o ponto de corte usando os valores máximos.

Por exemplo, se o desenho de uma imagem preencher somente a região inferior direita com quadrante de coordenadas (150, 150), (150, 200), (200, 150) e (200, 200), é calculada a variação máxima de (50, 50) para definir como o novo tamanho e é feita a translação de (-150, -150) já que (150, 150) eram as coordenadas mínimas.

No restante do código, mostrado à direita na Figura 56, é criada a imagem com um `PictureRecorder` aplicado no `Canvas`, criando um quadro branco com o método `drawRect` e desenhando os traços sobre o ele com o método `drawLine` a partir dos novos pontos transladados.

```
140                     setState(() {
141                       predictionBoxesList = [];
142                       for(var i = 0; i < predictions.length; i++){
143                         var predProbIndexKanji = predictions[i].values.toList();
144                         predictionBoxesList.add(
145                           predictionBox(predProbIndexKanji[1], predProbIndexKanji[2])
146                         );
147                       }
148                     });
```

Figura 57 - Inserção dos *widgets* da lista de *Kanjis* previstos para visualização no `Expanded` superior.

```
61      SizedBox predictionBox(int index, String kanji) {
62        return SizedBox(
63          height: 50,
64          width: 50,
65          child: Container(
66            margin: EdgeInsets.only(left: 2.0, right: 2.0),
67            child: FlatButton(
68              color: Colors.white, textColor: Colors.black, splashColor: Colors.blueAccent,
69              padding: EdgeInsets.all(2.0),
70              onPressed: () {
71                LinkedHashMap kanjiMap = kanjiTable[index];
72                Navigator.push(
73                  context,
74                  MaterialPageRoute(builder: (context) => PredictionPage(kanjiMap: kanjiMap)),
75                );
76              },
77              child: Text(kanji, style: TextStyle(fontSize: 30.0)),
78            ))); // FlatButton // Container // SizedBox
79      }
```

Figura 58 - Estruturação do *widget* customizado `predictionBox` contendo o *Kanji* previsto.

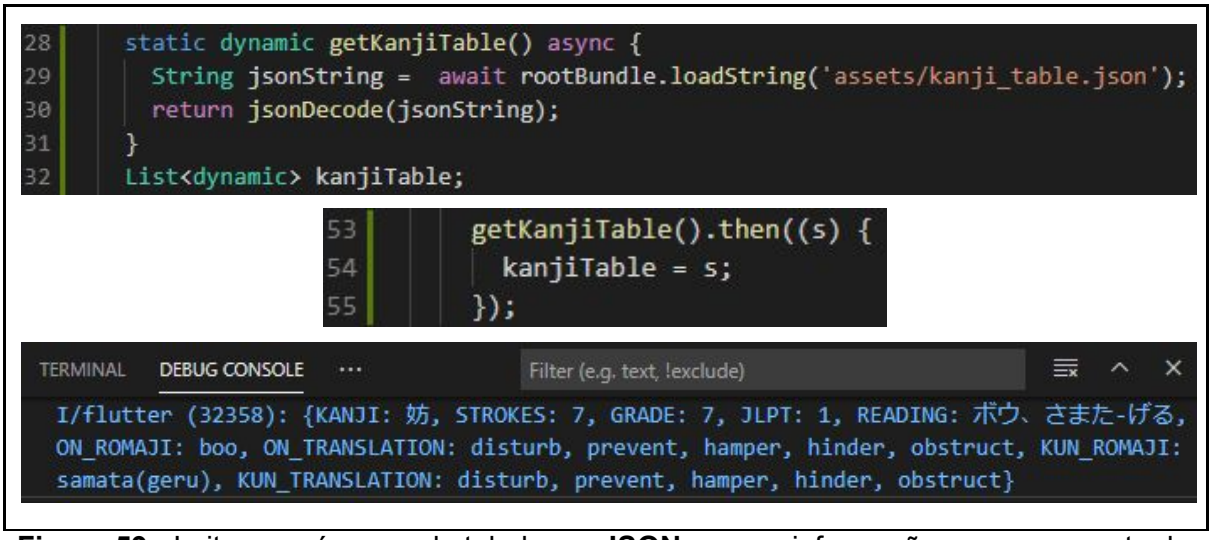


Figura 59 - Leitura assíncrona da tabela em **JSON** com as informações, carregamento da tabela no bloco `initState` e *print* destas informações referentes a um *Kanji*.

Obtida a lista de possíveis *Kanjis* previstos para a imagem, é feito um laço sobre ela na Figura 57 para criar uma lista `predictionBoxesList` com estes símbolos onde é possível o usuário clicar para obter informações sobre ele. Isso é feito com um método que cria um `SizedBox` customizado que foi chamado de predictionBox.

Suas dimensões foram definidas em 50x50 pixels. No evento de ser pressionado (parâmetro `onPressed` da linha 70, Figura 58), as informações da tabela de *Kanjis* relativas a seu índice são armazenadas na variável `kanjiMap`, a partir da tabela `kanjiTable`, gerada no `initState` quando lido o arquivo **JSON** (centro da Figura 59).

Usando o método `Navigator.push` (linha 72 da Figura 58), é possível fazer o aplicativo avançar uma página, passando para ele como parâmetro um novo *widget* global `PredictionPage`, que foi estruturado no último arquivo de código restante, o prediction_page**.dart**.

```
91  Expanded(
92    flex: 2,
93    child: Container(
94      padding: EdgeInsets.all(16),
95      color: Colors.black54,
96      alignment: Alignment.center,
97      child: Scrollbar(child:
98        SingleChildScrollView(
99          scrollDirection: Axis.horizontal,
100         child: Row(
101           children: predictionBoxesList
102         ), // Row
103       ), // SingleChildScrollView
104   ), // Scrollbar
105   ), // Container
106 ), // Expanded
```

Figura 60 - Código para o `Expanded` superior.

Figura 61 - Interface do `Expanded` superior no aplicativo.

A função de um *widget* `Expanded` é preencher todo o espaço que for possível na orientação do *widget* parental e de acordo com o valor que lhe foi dado no parâmetro `flex`. A visão da tela principal do aplicativo continha uma coluna com o `Expanded` superior, o `Container` no centro e o `Expanded` inferior.

O tamanho do `Container` é fixo, com 200x200 pixels para o canvas central, mais 40 pixels mínimos ao redor dele definidos na sua margem, totalizando 280 pixels de altura. Os dois `Expanded` restantes possuem `flex` 2 e 3 respectivamente. Isso significa que, como o `flex` total é 5, o primeiro `Expanded` ocupa 2/5 (dois quintos) da altura restante na tela, enquanto o segundo `Expanded` ocupa 3/5 (três quintos).

De acordo com as Figuras 60 e 61, o conteúdo deste Expanded é a lista de `SizedBoxes` dos *Kanjis* previstos. A lista é englobada pelo *widget* `SingleChildSchollView` para que permita ao usuário “deslizá-la” horizontalmente caso o comprimento da lista ultrapasse a largura da tela. Este, por sua vez, é englobado pelo *widget* `Scrollbar`, para que a barra de rolagem fique visível.

```
lib > prediction_page.dart > PredictionPage
4   class PredictionPage extends StatelessWidget {
5     PredictionPage({this.kanjiMap});
6     final LinkedHashMap kanjiMap;
7
8   >   Expanded infoCard(String title, String mapKey, {int flex = 1}) {···
24
25    @override
26    Widget build(BuildContext context) {
27
28      return Scaffold(
29        backgroundColor: Colors.black,
30        appBar: AppBar(
31          title: Text("Kanji " + kanjiMap['KANJI']),
32        ), // AppBar
33        body: SingleChildScrollView(
34          scrollDirection: Axis.vertical,
35          child: Column(
36            mainAxisAlignment: MainAxisAlignment.spaceAround,
37            children: [
38  >           Row( // Row
44  >           Row( // Row
52  >           Row( // Row
58  >           Row( // Row
64  >           Row( // Row
70  >           Row( // Row
76            ]
77          ) // Column
78        ), // SingleChildScrollView
79      ); // Scaffold
80
81    }
82  }
```

Figura 62 - *Widget* `PredictionPage`, código da segunda página do aplicativo.

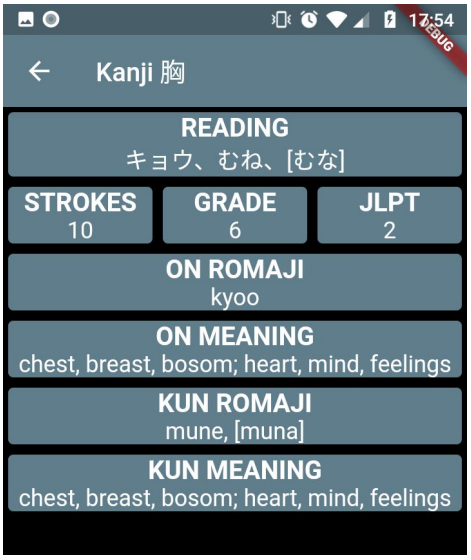


Figura 63 - Interface da segunda página do *app*.

Ao clicar no `SizedBox` contendo o *Kanji* desejado, é aberta a página com as informações originárias da tabela. O *widget* global desta página, `PredictionPage`,

é sem estado, já que ele é sempre criado do zero quando a caixa da página anterior é pressionada, não precisando armazenar estado.

O corpo do `Scaffold` em seu código da Figura 62 contém as 8 informações definidas na tabela para o *Kanji*: número de traços, ano em que é ensinado, em qual teste de proficiência cai, leituras em *kana*, leitura *onyomi*, tradução *onyomi*, leitura *kunyomi*, tradução *kunyomi*. Cada uma delas está estruturada em fileiras contendo pelo menos um `Card`, colocadas verticalmente e englobados por um `SingleChildScrollView` para permitir rolagem (desta vez vertical) caso ultrapasse os limites da tela.

```
8   Expanded infoCard(String title, String mapKey, {int flex = 1}) {
9     return Expanded(
10      flex: flex,
11      child: Card(
12        color: Colors.blueGrey,
13        child: Column(
14          children: [
15            Text(title, textAlign: TextAlign.center, style: TextStyle(
16              color: Colors.white, fontWeight: FontWeight.bold, fontSize: 20)), // TextStyle // Text
17            Text(kanjiMap[mapKey].toString(), textAlign: TextAlign.center, style: TextStyle(
18              color: Colors.white, fontSize: 18)) // TextStyle // Text
19          ]
20        ) // Column
21      ) // Card
22    ); // Expanded
23  }
```

Figura 64 - Método `infoCard` para criar um `Card` customizado.

Como visto na Figura 63, cada `Card` é composto pelo título em negrito e sua respectiva informação logo abaixo. Logo, dois parâmetros convenientes para definir na chamada do método da Figura 64 é uma *string* para o título (`title`) e outra *string* contendo o índice da coluna para acessar na lista de informações (`mapKey`).

Um terceiro parâmetro opcional foi pré-definido para o flex. Como o `Card` está englobado por um `Expanded`, `flex` de valor 1 faz o `Card` preencher completamente a fileira na horizontal. Se a fileira contiver três `Cards` de `flex` com valor 1, os três `Cards` vão dividir a fileira igualmente em três partes. Isso foi feito para colocar com facilidade as informações de traços, ano e teste de proficiência numa mesma fileira, já que cada informação contém somente um caractere. Exemplos de preenchimento destes `Cards` estão dispostos na Figura 65.

```
34        body: SingleChildScrollView(
35          scrollDirection: Axis.vertical,
36          child: Column(
37            mainAxisAlignment: MainAxisAlignment.spaceAround,
38            children: [
39              Row(
40                mainAxisAlignment: MainAxisAlignment.spaceBetween,
41                children: [
42                  infoCard('READING', 'READING'),
43                ]
44              ), // Row
45              Row(
46                mainAxisAlignment: MainAxisAlignment.spaceBetween,
47                children: [
48                  infoCard('STROKES', 'STROKES'),
49                  infoCard('GRADE', 'GRADE'),
50                  infoCard('JLPT', 'JLPT')
51                ]
52              ), // Row
53              Row(
54                mainAxisAlignment: MainAxisAlignment.spaceBetween,
55                children: [
56                  infoCard('ON ROMAJI', 'ON_ROMAJI'),
57                ]
58              ), // Row
```

Figura 65 - Preenchimento das fileiras pelos `infoCards` na coluna do `SingleChildScrollView`.

O terceiro e último *widget* no corpo do `Scaffold` da tela principal é o `Expanded` inferior, como no código da Figura 66.

```
163                 Expanded(
164                   flex: 3,
165                   child: Container(
166                     padding: EdgeInsets.all(16),
167                     color: Colors.black54,
168                     alignment: Alignment.center,
169                     child: Container(
170                       width: 48,
171                       height: 48,
172                       margin: EdgeInsets.all(40),
173                       decoration: new BoxDecoration(
174                         color: Colors.white,
175                         border: new Border.all(
176                           width: 1,
177                           color: Colors.black,
178                         ), // Border.all
179                       ), // BoxDecoration
180                       child: FittedBox(
181                         child: Container(
182                           child: brain.processedImg,
183                         ), // Container
184                         fit: BoxFit.fill,
185                       ), // FittedBox
186                     ), // Container
187                   ), // Container
188                 ), // Expanded
```

Figura 66 - Codificação do segundo `Expanded` da tela principal.

A única função do `Expanded` inferior é mostrar um *preview* da imagem que passará pelo processo de reconhecimento, com a remoção do espaço em branco envoltório e redimensionamento para 48x48, como na Figura 67. Este *preview* fica armazenado em `brain.processedImg`, e é atualizado a cada traço feito pelo usuário.

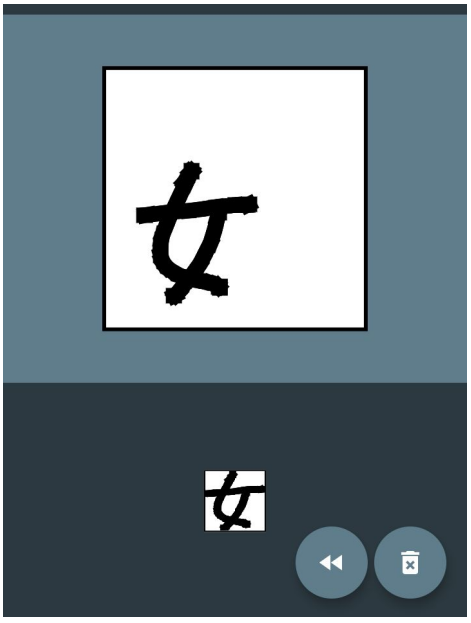


Figura 67 - Exemplo de *preview* da imagem a passar pelo reconhecedor, com o envoltório em branco removido.

Por fim, a única parte restante do código é referente aos dois botões flutuantes, um para “rebobinar” os tracejados e outro para limpar completamente o canvas.

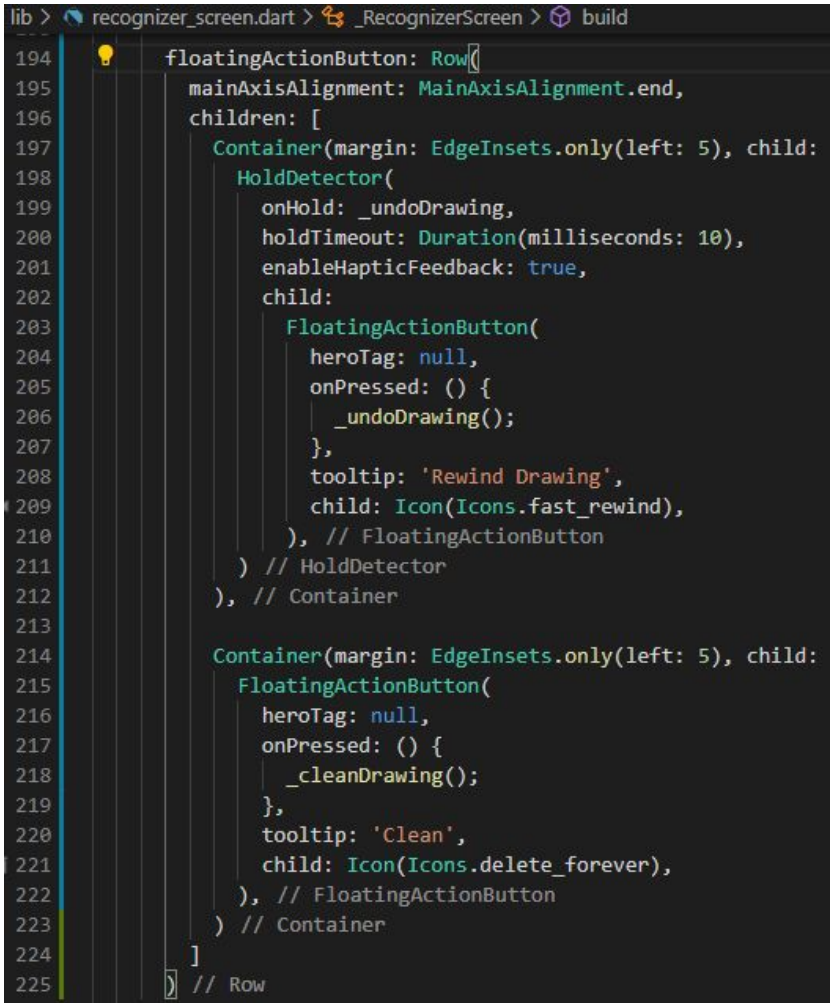
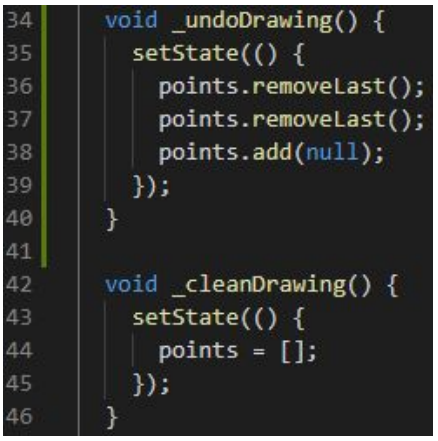


Figura 68 - Código para os dois botões flutuantes da tela principal.

Para poder usar dois botões flutuantes, diferentemente do aplicativo *template*, o parâmetro `floatingActionButton` na Figura 68 recebe uma `Row` contendo dois `Containers`, um para cada botão.

Outra alteração necessária para a utilização de mais de um botão é definir suas `heroTags` como nulas. A propriedade ‘hero’ tem relação com animações (não utilizadas neste caso), e por padrão Flutter sempre aloca a mesma `heroTag` para um `floatingActionButton`, gerando erros de execução quando há dois botões.



```
34    void _undoDrawing() {
35      setState(() {
36        points.removeLast();
37        points.removeLast();
38        points.add(null);
39      });
40    }
41
42    void _cleanDrawing() {
43      setState(() {
44        points = [];
45      });
46    }
```

Figura 69 - Definição para as funções realizadas em cada botão.

O primeiro botão tem função de “rebobinar” o traço quando pressionado continuamente. Para alocar uma função que seja executada repetidamente enquanto o botão for mantido apertado, foi preciso usar o *widget* `HoldDectector`, presente no pacote externo holding_gesture, importado previamente.

Como definido na linha 200 da Figura 68, a cada 10 milissegundos de tempo em que o botão estiver pressionado será executada a função `_undoDrawing` da Figura 69. Em sua execução, é necessário remover os dois últimos pontos da lista `points`, já que é preciso acrescentar um ponto nulo para que o canvas não interligue traços erroneamente, e removendo só um ponto enquanto é adicionado outro não faz o traço voltar no tempo.

Para o segundo botão, basta usar um `floatingActionButton` comum que quando pressionado uma vez seja esvaziada a lista de pontos coletados, como feito na função `_cleanDrawing` da Figura 69.

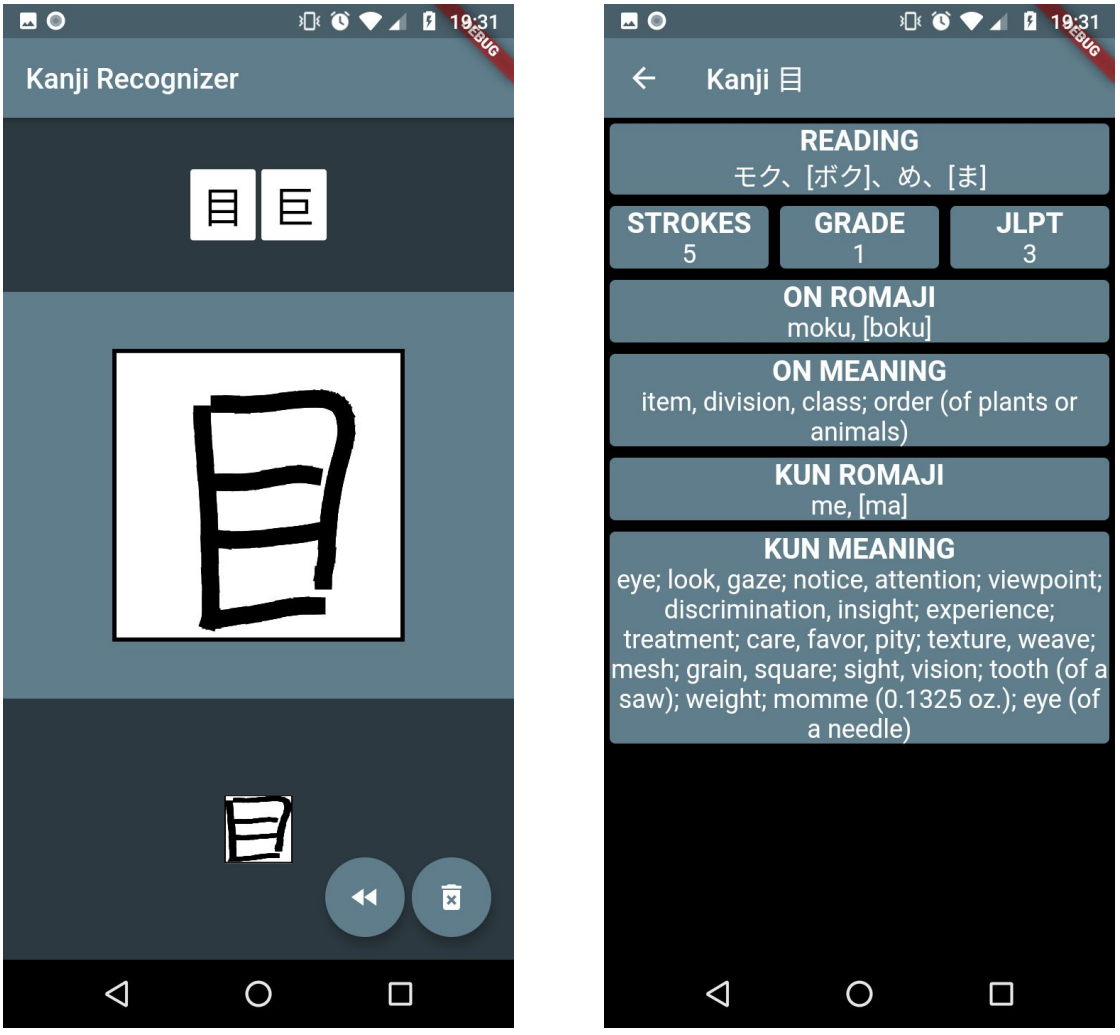


Figura 70 - As duas telas da versão final do aplicativo.

Com as telas do aplicativo finalizadas como na Figura 70, a última coisa a se fazer é gerar um executável no *smartphone* que rode o aplicativo sem necessidade de estar executando projeto no computador. Para uso apenas particular, sem necessidade de publicação na Play Store, uma maneira de se fazer isso é gerando um arquivo **APK** pelo terminal. No próprio VS Code, abrindo o terminal, se o diretório já estiver na pasta do projeto, basta executar 'flutter build apk'.

O processo termina após alguns minutos. Depois disso, mantendo o terminal aberto no mesmo diretório e conectando o smartphone em que se quer instalar o *app* via USB, basta executar no terminal 'flutter install'. Ele instala a versão *debug* com a faixa no canto superior direito, mas mantém a funcionalidade. Como pode ser visto no ícone da Figura 71, o nome do *app* é o mesmo da pasta do projeto criado: 'kanji_recognizer'.

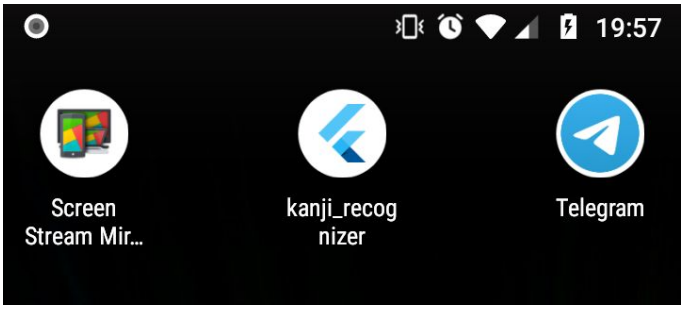


Figura 71 - Ícone do aplicativo na tela do celular.

5 - CONSIDERAÇÕES FINAIS

A ideia do projeto exigiu uma abordagem mais prática do que teórica. Apesar de terem sidos necessários fundamentação teórica em redes neurais, domínio em Python e conhecimento de Flutter e de design de interfaces, o documento não possui extensivos testes estatísticos e conclusões científicas. O projeto serve como guia de como pôr em prática vários conceitos em aprendizado de máquina e desenvolvimento de apps.

O banco de imagens foi criado artificialmente, e não representa exatamente amostras reais usadas no reconhecedor criado, que são os desenhos na tela do *smartphone*. Um método ideal para validação final do classificador seria coletando imagens desenhadas por vários usuários diferentes, aplicando o corte do envoltório em branco, redimensionamento em 48x48 e passagem pelo grafo para determinação da acurácia real. Esta etapa não foi possível de ser realizada por limitações de tempo, porém o uso particular e contínuo do aplicativo por uma pessoa foi julgado suficiente para concluir que o aplicativo tem alto poder de acurácia, suportado pelo fato de que tanto as imagens artificiais como as desenhadas no aplicativo passam por etapas de pré-processamento parecidas e possuem largura de traços similares, além das 35 fontes incluírem algumas feitas a mão.

6 - REFERÊNCIAS BIBLIOGRÁFICAS

Jōyō kanji - Wikipedia. Disponível em: <https://en.wikipedia.org/wiki/J%C5%8Dy%C5%8D_kanji>. Acesso em 30 de outubro de 2020.

DESHPANDE, Adit. A Beginner's Guide To Understanding Convolutional Neural Networks. Disponível em: <https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/>. Acesso em 30 de outubro de 2020.

NIELSEN, Michael. Neural networks and deep learning. Determination Press, 2015. Disponível em: <http://neuralnetworksanddeeplearning.com/>. Acesso em 11 de novembro de 2020.

RMSprop - Keras. Disponível em: <https://keras.io/api/optimizers/rmsprop>. Acesso em 11 de novembro de 2020.

Adam - Keras. Disponível em: <https://keras.io/api/optimizers/adam/>. Acesso em 11 de novembro de 2020.

CLOW, Mark. Learn Google Flutter Fast - 65 Example Apps. Publicação independente, 2019.

Visual Studio Code - Flutter. Disponível em: <https://flutter.dev/docs/development/tools/vs-code>. Acesso em 11 de novembro de 2020.

heroTag property - FloatingActionButton class. Disponível em: <https://api.flutter.dev/flutter/material/FloatingActionButton/heroTag.html>. Acesso em 15 de novembro de 2020.

Build and release an Android app - Flutter. Disponível em: <https://flutter.dev/docs/deployment/android#adding-a-launcher-icon>. Acesso em 15 de novembro de 2020.