

UNIVERSIDADE DA BEIRA INTERIOR

MESTRADO EM ENGENHARIA INFORMÁTICA

APRENDIZAGEM AUTOMÁTICA

---

*Deep Learning*

---

*Autora:*  
Rita CORREIA, M9933

*Professor Dr.:*  
Hugo PROENÇA



Julho, 2020

# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>6</b>
1.1	Motivação e Contexto . . . . .	6
1.2	Objetivos . . . . .	6
<b>2</b>	<b>Redes Neuronais Convolucionais</b>	<b>7</b>
<b>3</b>	<b>Desenvolvimento e implementação</b>	<b>11</b>
3.1	IDE e linguagem . . . . .	11
3.2	CPU vs GPU . . . . .	11
3.3	<i>Dataset</i> . . . . .	12
3.4	<i>Imports</i> . . . . .	14
3.5	Separação do conjunto de treino e teste . . . . .	15
3.6	Armazenamento das imagens . . . . .	16
3.7	Processamento das imagens e classes . . . . .	17
3.8	Divisão do conjunto de treino e validação . . . . .	19
3.9	Arquitetura do modelo <i>Convolutional Neural Network</i> (CNN) . .	20
3.10	Normalização . . . . .	23
3.11	Treino do modelo . . . . .	25
3.12	Teste do modelo . . . . .	26
<b>4</b>	<b>Resultados obtidos e Testes experimentais</b>	<b>28</b>
4.1	Resultados obtidos . . . . .	28
4.2	Testes experimentais . . . . .	29
4.2.1	Menor número de iterações . . . . .	29
4.2.2	Maior número de iterações . . . . .	30
4.2.3	Menor <i>batch size</i> . . . . .	32
4.2.4	Maior <i>batch size</i> . . . . .	33
4.2.5	Otimizador <i>Adaptive Moment Estimation</i> (Adam) . . . .	34
4.2.6	Otimizador <i>Stochastic Gradient Descent</i> (SGD) . . . . .	35
<b>5</b>	<b>Conclusão e trabalho futuro</b>	<b>37</b>

# Listas de Figuras

2.1	Bart - classe 1.	8
2.2	Homer - classe 2.	8
2.3	Matriz de características.	8
2.4	Exemplo de CNN.	8
2.5	Rede convolucional.	9
2.6	Operação de convolução.	10
3.1	Utilização da <i>Graphics Processing Unit</i> (GPU) no treino do modelo.	12
3.2	Sinal de perigo - exemplo 1.	13
3.3	Sinal de perigo - exemplo 2.	13
3.4	Sinal de perigo - exemplo 3.	13
3.5	Sinal de perigo - exemplo 4.	13
3.6	Sinal de informação - exemplo 1.	14
3.7	Sinal de informação - exemplo 2.	14
3.8	Sinal de informação - exemplo 3.	14
3.9	Sinal de informação - exemplo 4.	14
3.10	Algumas imagens armazenadas.	16
3.11	Representação da primeira imagem.	18
3.12	Classes das imagens.	18
3.13	Visualização das primeiras 5 imagens.	18
3.14	Visualização do número de classes.	19
3.15	<i>Shape</i> de X e y.	19
3.16	<i>Shape</i> de X e y após a divisão dos conjuntos de teste e validação.	20
3.17	Camada <i>Flatten</i> .	22
3.18	Sumário do modelo.	23
3.19	Treino do modelo no terminal.	25
4.1	<i>Accuracy</i> do treino e validação.	28
4.2	Custo do treino e validação.	28
4.3	Figuras 50 à 65 - resultados obtidos.	29
4.4	<i>Accuracy</i> do treino e validação - 20 iterações.	30
4.5	Custo do treino e validação - 20 iterações.	30
4.6	Imagen 50 à 65 - 20 iterações.	30
4.7	<i>Accuracy</i> do treino e validação - 200 iterações.	31
4.8	Custo do treino e validação - 200 iterações.	31

4.9	Imagen 150 à 165 - 200 iterações.	31
4.10	<i>Accuracy</i> do treino e validação - <i>batch size</i> de 10.	32
4.11	Custo do treino e validação - <i>batch size</i> de 10.	32
4.12	Imagen 304 à 319 - <i>batch size</i> de 10.	32
4.13	<i>Accuracy</i> do treino e validação - <i>batch size</i> de 100.	33
4.14	Custo do treino e validação - <i>batch size</i> de 100.	33
4.15	Imagen 304 à 319 - <i>batch size</i> de 100.	34
4.16	<i>Accuracy</i> do treino e validação - otimizador Adam.	34
4.17	Custo do treino e validação - otimizador Adam.	34
4.18	Imagen 185 à 200 - otimizador Adam.	35
4.19	<i>Accuracy</i> do treino e validação - otimizador SGD.	36
4.20	Custo do treino e validação - otimizador SGD.	36
4.21	Imagen 215 à 230 - otimizador SGD.	36

# Listings

3.1	Separação das imagens para o conjunto de teste. . . . .	15
3.2	Armazenamento das imagens. . . . .	16
3.3	Leitura e processamento das imagens. . . . .	17
3.4	Divisão em conjunto de treino e validação. . . . .	20
3.5	Arquitetura do modelo CNN. . . . .	20
3.6	Compilação do modelo. . . . .	22
3.7	Normalização das imagens. . . . .	24
3.8	Criação dos geradores. . . . .	24
3.9	Treino do modelo. . . . .	25
3.10	Extração das métricas no treino do modelo. . . . .	26
3.11	Pré-processamento no conjunto de teste. . . . .	26
3.12	Previsões e <i>display</i> das imagens de teste. . . . .	26

# Acrónimos

<b>Adam</b>	<i>Adaptive Moment Estimation</i>
<b>CNN</b>	<i>Convolutional Neural Network</i>
<b>CPU</b>	<i>Central Process Unit</i>
<b>GPU</b>	<i>Graphics Processing Unit</i>
<b>IA</b>	Inteligência Artificial
<b>IDE</b>	<i>Integrated Development Environment</i>
<b>ReLU</b>	<i>Rectified Linear Unit</i>
<b>RGB</b>	<i>Red, Green, Blue</i>
<b>RMSProp</b>	<i>Root Mean Square Propagation</i>
<b>SGD</b>	<i>Stochastic Gradient Descent</i>
<b>TI</b>	Tecnologias de Informação
<b>UBI</b>	Universidade da Beira Interior
<b>UC</b>	Unidade Curricular

# Capítulo 1

## Introdução

### 1.1 Motivação e Contexto

A área de *Deep Learning* (Aprendizagem Profunda) está relacionada à aplicação das redes neurais artificiais na resolução de problemas complexos e que requeiram poder computacional mais avançado. Existem diversas aplicações práticas que já foram construídas utilizando essas técnicas, tais como: carros autónomos, descoberta de novos medicamentos, diagnóstico antecipado de doenças, reconhecimento facial, previsão de valores na bolsa, etc. Nesses exemplos, a técnica base utilizada são as redes neurais artificiais (principalmente as CNN), que procuram simular o modo de funcionamento do cérebro humano. Por este motivo, é de extrema importância que os profissionais de Tecnologias de Informação (TI) saibam trabalhar com estas ferramentas, já que grandes empresas as utilizam nos seus sistemas, empresas estas como: Airbus, eBay, Dropbox, Intel, IBM, Uber, Twitter, Snapchat e também o próprio Google.

Selecionar e implementar um problema prático onde um classificador de *Deep Learning* possa ser utilizado para distinguir elementos de diferentes classes, foi o desafio proposto neste projeto aos alunos que frequentam a Unidade Curricular (UC) de Aprendizagem Automática presente no 2ºsemestre do Mestrado em Engenharia Informática, na Universidade da Beira Interior (UBI).

### 1.2 Objetivos

O presente projeto tem como principais objetivos:

- Adquirir manualmente um *dataset* composto por, no mínimo, 1000 fotografias com a mesma dimensão e espectro de cores. Cada imagem deve ser capturada com a maior variabilidade de fatores possíveis;
- Treinar e testar um modelo CNN apropriado sob o *dataset* escolhido;
- Realizar uma validação empírica do modelo com um conjunto de teste.

## Capítulo 2

# Redes Neuronais Convolucionais

As redes neurais convolucionais, inicialmente propostas por Yann LeCun em 1988, são o modelo clássico e mais popular de *Deep Learning*, sendo que foram desenvolvidas tomando como base o córtex visual de animais, que é composto por milhões de agrupamentos celulares complexos, sensíveis a pequenas sub-regiões do campo visual, chamadas de campos recetíveis. Essas regiões, nas redes neurais convolucionais, são também denominadas de campos recetivos, e são formadas por subconjuntos selecionados do vetor de características (representação numérica do objeto) a ser analisado [1].

As CNN são tipicamente usadas em Visão Computacional, subárea da Inteligência Artificial (IA) que se preocupa com o processamento de vídeos e imagens (por exemplo em projetos de robótica com reconhecimento de objetos), e que visa simular em computadores a visão que o Ser Humano tem do Mundo.

No livro "*Neural Networks and Learning Machines*"[3], Simon Haykin divide a estrutura das redes convolucionais em três objetivos principais:

- **Extração de características:** cada neurônio recebe sinais de entrada de um campo recetível da camada anterior, possibilitando a extração de características locais. Esta extração de características locais faz com que a posição exata de cada característica (ou pixel, no caso de uma imagem), seja irrelevante, desde que sua posição em relação às características vizinhas seja mantida [1].

De uma forma mais simples, a extração de características vai verificar e selecionar automaticamente quais são as características principais de cada classe (podem ser, por exemplo, cores ou formas), não utilizando assim todos os pixels de cada classe (ou imagem, neste caso). De um modo exemplificativo podemos considerar que temos duas personagens, o Bart (figura 2.1) e o Homer (figura 2.2), sendo que cada uma representa, neste caso, uma classe distinta. Caracterizando ambas as classes de acordo com as suas características próprias, podemos verificar que o Bart tem uma

camisola laranja, calções azuis e sapatos azuis. Já o Homer tem camisola branca, calças azuis (outro valor de azul) e sapatos cinzentos escuros.



Figura 2.1: Bart - classe 1.



Figura 2.2: Homer - classe 2.

Na imagem 2.3 temos um exemplo de uma matriz de características, para o caso das duas classes anteriormente referidas, onde é possível encontrar as características da classe Bart nos 3 primeiros valores, seguidos dos valores correspondentes às características da classe Homer, e por fim a classe a que cada imagem pertence. Estes valores são assim extraídos utilizando uma CNN (figura 2.4) sendo que nesta, os pontos vermelhos (*input layer*) são as características totais (tanto da classe Bart como da Homer), os neurónios (representados por pontos) amarelos são as camadas escondidas (onde é efetuado o processamento das imagens e treino do modelo) e os neurónios azuis são relativos à camada de saída (neste caso só precisaríamos de dois, pois só temos 2 classes).

```
8.97,3.45,2.35,0.0,00.00,0.00,Bart
6.75,0.94,0.52,0.00,0.00,0.00,Bart
9.69,4.10,1.56,0.00,0.00,0.00,Bart
0.00,0.00,0.00,4.68,0.66,0.01,Homer
0.00,0.00,0.00,0.12,2.50,0.03,Homer
0.00,0.00,0.00,5.80,0.50,1.28,Homer
```

Figura 2.3: Matriz de características.

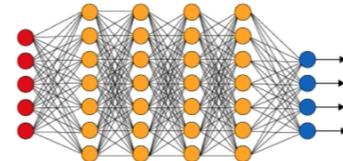


Figura 2.4: Exemplo de CNN.

- **Mapeamento de características:** cada camada da rede é composta por diversas matrizes de características (*feature maps*), que são regiões onde os neurónios compartilham os mesmos pesos sinápticos. Esses pesos são chamados de filtros, e dão robustez ao modelo, fazendo com que este seja capaz de lidar com variações de distorção, rotação e translação na imagem. O compartilhamento dos pesos também possibilita uma redução enorme no número de parâmetros a serem otimizados [1].

- **Subamostragem:** após cada camada de convolução é aplicada uma camada de subamostragem (*subsampling*), que nada mais é do que uma coleção de amostras de cada matriz de características. Estas amostragens podem ser realizadas obtendo-se a soma, tirando a média, selecionando-se o maior (*max pooling*) ou menor (*min pooling*) valor da região em análise, o que produz uma summarização desta [1].

A imagem 2.5, extraída do livro de Haykin [3], mostra uma rede convolucional aplicada numa imagem de  $28 \times 28$  pixels, colocando-se na primeira camada quatro filtros de tamanho  $24 \times 24$ , e em seguida realizando-se uma subamostragem. O processo segue com a aplicação de novos filtros e subamostragens.

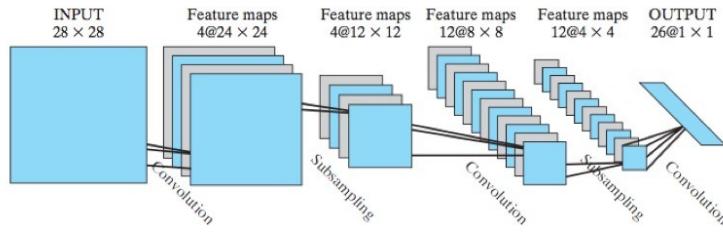


Figura 2.5: Rede convolucional.

Na figura 2.6 temos um exemplo de uma imagem representada por uma matriz binária (podemos considerar uma imagem em preto e branco), onde vamos aplicar o que seria uma camada da rede, ou seja, realizar a convolução e *subsampling*. Neste caso, e para tornar o exemplo simples de entender, aplicamos apenas um filtro representado pela matriz ilustrada a amarelo.

Podemos observar que cada região a verde é uma matriz de características, que é obtida "deslizando-se" o filtro  $3 \times 3$  pela imagem. Em cada matriz de características é aplicado o filtro em amarelo que, na verdade, é uma multiplicação matricial (matriz X filtro). Esta operação é denominada de **convolução**. Por fim, para obter cada valor da nova matriz  $6 \times 6$ , realizamos um *max pooling*, ou seja, selecionamos o maior valor da matriz resultante da aplicação do filtro na matriz de características [1].

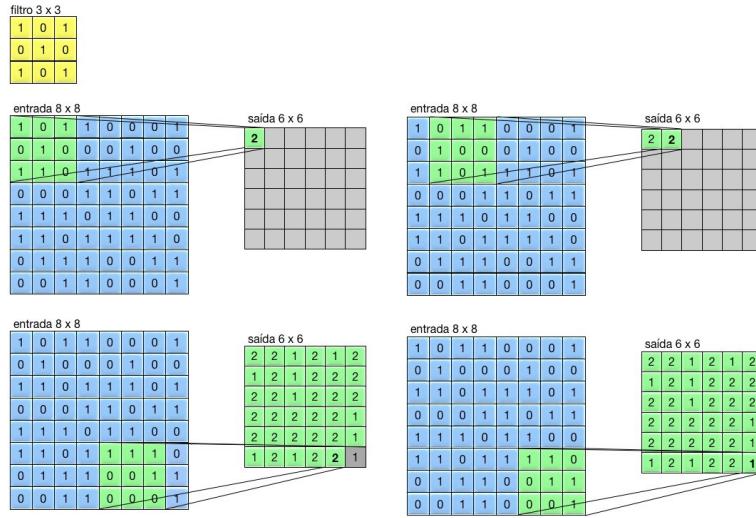


Figura 2.6: Operação de convolução.

Os valores dos filtros são, portanto, os pesos da rede. Esses pesos são aprendidos durante o processo de treino, geralmente utilizando-se a técnica de retropropagação (*backpropagation*).

Para problemas de classificação, na última camada da rede podemos aplicar uma função *softmax*, afim de obtermos a probabilidade de dada imagem pertencer a qualquer uma das possíveis classes presentes no problema [1].

## Capítulo 3

# Desenvolvimento e implementação

### 3.1 IDE e linguagem

O projeto foi desenvolvido no *Integrated Development Environment* (IDE) *Visual Studio Code*. Este *software* é um editor de texto que oferece um conjunto vasto de funcionalidades. É também multi-plataforma, gratuito e direcionado para quem procura um simples IDE , leve mas ao mesmo tempo bastante completo e com um vasto leque de extensões. A linguagem de programação utilizada foi o Python, na versão 3.6.10, sob a distribuição Anaconda.

### 3.2 CPU vs GPU

Inicialmente estava a ser utilizado o *Central Process Unit* (CPU) da máquina local para o processamento do treino do modelo criado através da CNN. No entanto, cada *epoch* (iteração) demorava mais de 40 segundos a ser executada, o que dá um total de aproximadamente 1h e meia para 100 iterações. Desta forma, o processo de desenvolvimento, teste e *debug* era um processo super lento e moroso.

Por este motivo, e com a distribuição do Anaconda, foi utilizada a GPU, criando um ambiente virtual "PythonGPU"e instalando lá as dependências do tensorflow para este efeito (*tensorflow-gpu*). Foi ainda necessário descarregar e instalar a *toolkit CUDA*, que é uma API da NVIDIA que fornece um ambiente de desenvolvimento para criar aplicações com alta performance a executar sobre a GPU. Este kit de ferramentas inclui bibliotecas para uso de CPU, ferramentas de *debugging* e otimização, um compilador C/C++ e uma biblioteca de tempo de execução [4].

Na figura 3.1 conseguimos observar o uso da GPU aquando o treino do modelo.

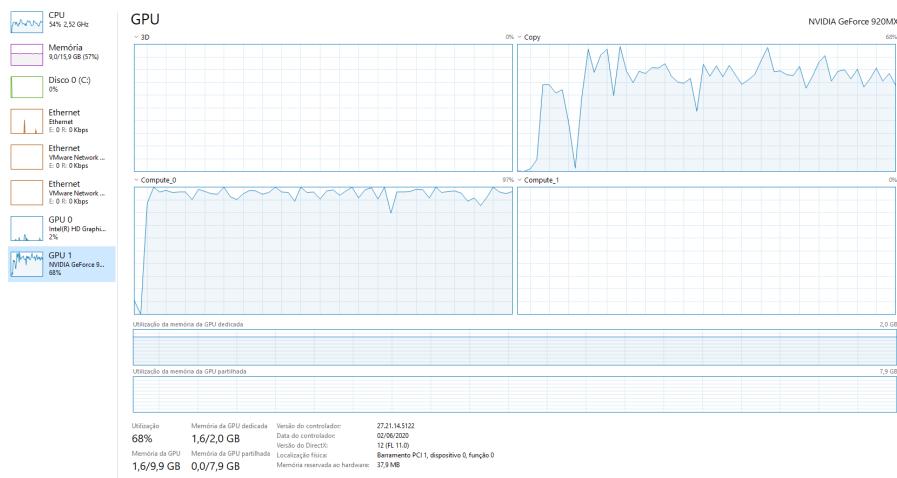


Figura 3.1: Utilização da GPU no treino do modelo.

### 3.3 Dataset

O problema que explorei neste projeto consiste na identificação e distinção entre duas classes compostas por sinais de trânsito de informação e de perigo, respectivamente.

Todas as imagens foram capturadas com o mesmo dispositivo móvel, estando num mesmo formato (.jpg), num mesmo espectro de cores (*Red, Green, Blue* (RGB)) e com apenas uma instância do objeto a distinguir por imagem. As fotografias foram também capturadas em diferentes alturas do dia (desde de manhã à noite), em diferentes perspetivas e distâncias, com diferentes graus de nitidez, tendo também sido alguns sinais nas imagens parcialmente cortados.

O *dataset* conta com um total de 1604 fotografias, sendo que 320 destas (cerca de 20% do total) se encontram no conjunto de teste, existindo neste o mesmo número de fotografias de ambas as classes de sinais. As restantes 1284 estão presentes no conjunto de treino, cerca que cerca de 20% destas foram utilizadas para validação.

Nas figuras 3.2 à 3.6 estão representadas alguns exemplos de imagens, referentes a ambas as classes, presentes no *dataset*, retiradas tanto dos conjuntos de treino como de teste.



Figura 3.2: Sinal de perigo - exemplo 1. Figura 3.3: Sinal de perigo - exemplo 2.



Figura 3.4: Sinal de perigo - exemplo 3. Figura 3.5: Sinal de perigo - exemplo 4.



Figura 3.6: Sinal de informação - exemplo 1.



Figura 3.7: Sinal de informação - exemplo 2.



Figura 3.8: Sinal de informação - exemplo 3.



Figura 3.9: Sinal de informação - exemplo 4.

### 3.4 Imports

Para a implementação do programa foi necessário fazer diversos *imports* (um total de 19), nomeadamente:

- **cv2:** também chamada OpenCV, é uma biblioteca de processamento de imagem e vídeo disponível em Python e em muitas outras linguagens de programação de alto nível. É usado para todos os tipos de análise de imagem e vídeo, como reconhecimento e detecção facial, leitura de placas, edição de fotos, visão robótica avançada, reconhecimento óptico de caracteres e muito mais. Foi utilizada no projeto para efetuar a leitura e redimensionamento das imagens;

- **matplotlib.pyplot e matplotlib.image**: o ".pyplot" é uma biblioteca de *plotting* para o Python, que pode ser usada para traçar linhas, gráficos e histogramas. Com o módulo ".image" é possível mostrar também imagens;
- **os**: é um módulo embutido do python para aceder ao computador e ao sistema de arquivos. Pode ser usado para exibir conteúdo em diretórios, criar novas pastas e excluir pastas;
- **gc**: abreviação de *garbage collector*, é uma ferramenta importante para limpar e excluir manualmente variáveis desnecessárias;
- **train \_ test \_ split**: é um método do sklearn.model\_selection e foi utilizado para dividir o conjunto de treino e validação;
- **layers**: é uma classe da biblioteca **Keras** que contém diferentes tipos de *layers* usados em *deep learning*, tais como: *Convolutional layer* (o mais utilizado em visão computacional), *Pooling layer*, *Recurrent layer*, *Embedding layers* (geralmente usado no processamento de linguagem natural), *Normalization layers*, entre outros;
- **models**: é uma classe da biblioteca **Keras** que contém dois tipos de modelos: *Sequential model* (o que utilizamos neste projeto) e o modelo com API funcional;
- **optimizers**: é um módulo da biblioteca **Keras** que contém vários otimizadores, ou seja, tipos de algoritmos de retro propagação, para treinar o modelo, tais como: *Root Mean Square Propagation* (RMSProp), SGD, Adam, adagrad, adadelta, etc;
- **ImageDataGenerator**: é uma classe da biblioteca **Keras** que é utilizada quando estamos a trabalhar com *datasets* de tamanhos reduzidos.

### 3.5 Separação do conjunto de treino e teste

Para a separação do *dataset* nos conjuntos de treino e teste foram, inicialmente, criadas duas variáveis "*train\_dir*" e "*test\_dir*" que contém o *path* para as diretórios de treino e teste, respetivamente. De seguida, e com uma *seed* de 3, foram selecionados de forma pseudo-aleatória, 160 imagens de sinais de informação (20% do total de sinais de informação existentes) e 160 imagens de sinais de perigo, como podemos ver no excerto 3.1.

```
random.seed(3)
#sinais de info para teste
info_moved = random.sample(glob.glob("C:\\\\Users\\\\ritac\\\\Desktop\\\\
    TP5_ML\\\\train\\\\train\\\\info *.jpg"), 160)
#sinais de perigo para teste
perigo_moved = random.sample(glob.glob("C:\\\\Users\\\\ritac\\\\Desktop\\\\
    TP5_ML\\\\train\\\\train\\\\perigo *.jpg"), 160)
```

Listing 3.1: Separação das imagens para o conjunto de teste.

Se a diretoria de teste estiver vazia, vai mover as fotos previamente selecionadas de ambas as classes para o conjunto de teste, através do método `move()` da biblioteca `shutil`.

### 3.6 Armazenamento das imagens

Como podemos ver no excerto 3.2, o programa vai encontrar todas as imagens de treino e informação, no conjunto de treino, guardando-as em 2 variáveis distintas (`train_info` e `train_perigo`). As imagens no conjunto de teste irão também ser guardadas numa variável denominada de `test_imgs`.

```
# sinais info
train_info = [ 'train/train/{}'.format(i) for i in os.listdir(
    train_dir) if 'info' in i] #tdas as img c a palavra info (642
    fotos)
# sinais perigo
train_perigo = [ 'train/train/{}'.format(i) for i in os.listdir(
    train_dir) if 'perigo' in i] #tdas as img c a palavra perigo
    (642 fotos)
# todas as imgs de teste
test_imgs = [ 'test/test/{}'.format(i) for i in os.listdir(test_dir)]
    ]
```

Listing 3.2: Armazenamento das imagens.

De seguida, as imagens de informação e perigo no conjunto de treino são armazenadas numa só, denominada de `train_imgs`, e ambos os conjuntos de treino e teste aleatorizam a ordem das fotografias, através do método `shuffle()` da biblioteca `random`. Para não ficarmos sem memória na altura de treinar o modelo, fazemos também uma limpeza às variáveis que já não são necessárias (`train_info` e `train_perigo`) através do método `collect()` da biblioteca `gc`. Para verificar se as 4 primeiras imagens estavam a ser passadas para o programa de forma correta, foi dado um pequeno `plot` às mesmas utilizando o módulo `mpimg`, como podemos observar na figura 3.10.

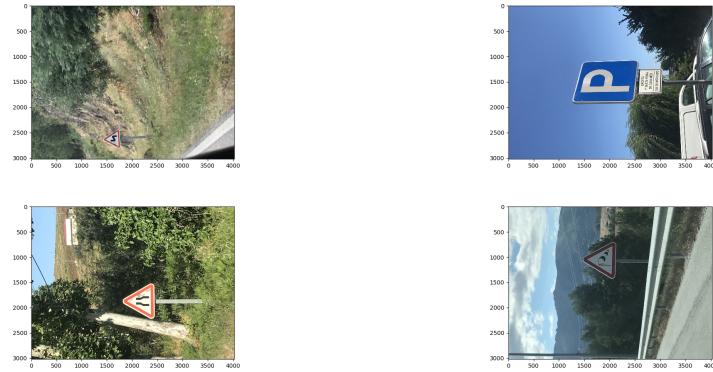


Figura 3.10: Algumas imagens armazenadas.

### 3.7 Processamento das imagens e classes

Nesta secção irá ser explicado como foi feito o processamento das imagens armazenadas na secção anterior. Para este efeito, foram inicialmente declaradas as variáveis que vão conter as dimensões a aplicar em todas as imagens. As variáveis "nrows" e "ncolumns", ambas com valor de 150, determinam a altura e largura das imagens, respetivamente, e a variável `channels` com valor igual a 3, determina o número de canais nas imagens. Uma imagem colorida é constituída por 3 canais correspondentes aos pixels vermelhos, verdes e azuis (RGB). Poderíamos também usar 1 canal que iria, nesse caso, ler as imagens no formato *gray-scale*, ou seja, a preto e branco.

Para ler e processar as imagens com as dimensões anteriormente mencionadas, foi criada uma função `read_and_process(images)`, apresentada no excerto 3.3.

```
def read_and_process(images):
    X = [] #imagens
    y = [] #labels
    for i in images:
        X.append(cv2.imread(i, cv2.IMREAD_COLOR, (nrows, ncolumns), interpolation = cv2.INTER_CUBIC))
        if 'info' in i:
            y.append(1)
        elif 'perigo' in i:
            y.append(0)
    return X,y
```

Listing 3.3: Leitura e processamento das imagens.

A função apresentada irá retornar duas variáveis: X e y. A variável X vai conter o conjunto de treino, enquanto a variável y vai conter as *labels*. De seguida, com a biblioteca do OpenCV (cv2), vamos ler e dimensionar todas as imagens com os valores previamente definidos, guardando-as no array X. Por último preenchemos o vetor y com as labels. Se a imagem contiver a palavra "info" no nome, vai ser adicionado o valor "1" ao array, caso contrário, 0.

Para verificarmos se as imagens estão de facto a ser processadas, e após aplicar a função anterior, fizemos um `print()` à primeira posição do X e ao y, tendo sido obtidos os resultados apresentados nas figuras 3.11 e 3.12.

```
[base]: C:\Users\ritac\Desktop\TP5_M\>c:/Users/ritac/anaconda3/python.exe c:/Users/ritac/Desktop/TP5_M/deep_learning_signs.py  
Conjunto de teste já existente! :)  
[[[ 89 109 92]  
 [ 76 97 82]  
 [ 74 93 74]  
 ...  
 [ 74 87 73]  
 [ 92 111 94]  
 [ 79 99 88]]]  
  
[[105 127 109]  
 [109 132 117]  
 [104 126 108]  
 ...  
 [ 57 66 53]  
 [ 68 87 66]  
 [ 81 100 81]]]  
  
[[ 97 116 97]  
 [ 73 92 73]  
 [ 89 108 91]  
 ...  
 [ 77 88 72]  
 [ 84 104 86]  
 [ 68 107 96]]]  
  
...  
  
[[ 15 15 9]  
 [ 14 14 8]  
 [ 15 15 9]  
 ...  
 [107 148 159]  
 [109 145 161]  
 [110 146 162]]]  
  
[[ 12 12 6]  
 [ 13 13 5]  
 [ 14 14 6]  
 ...  
 [108 150 154]  
 [109 147 157]  
 [110 148 163]]]
```

Figura 3.11: Representação da primeira imagem.

Figura 3.12: Classes das imagens.

Desta forma, foi possível entender como o computador lê cada imagem (pixel por pixel), chegando assim à conclusão que na primeira imagem se trata de um sinal de perigo (pois a *label* na primeira posição é 0). Para verificarmos o estado das imagens após o processamento destas, foi dado *plot* às 5 primeiras figuras presentes no vetor X, como podemos observar na figura 3.13. Neste caso, não é possível mostrar estas imagens com o módulo `mpimg` da biblioteca `matplotlib` pois estas são agora interpretadas como *arrays* de pixeis e não ficheiros `.jpg` puros. Assim sendo, foi necessário utilizar o módulo `imshow()`.

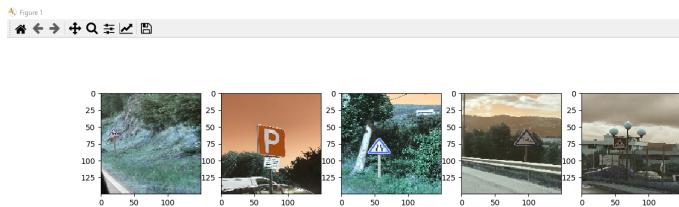


Figura 3.13: Visualização das primeiras 5 imagens.

Seguidamente, procedeu-se a uma nova limpeza de memória, e alterou-se as variáveis X e y, que estavam em formato *list*, para um *array numpy*, de forma a conseguir-mos inseri-las no modelo. Após ter a certeza que as imagens estavam a ser corretamente processadas e armazenadas, foi também prudente verificar as *labels/classes*. Se as ambas as classes estiverem a ser bem armazenadas no respetivo vetor (y), terão de existir 642 zeros e 642 uns, ou seja, 642 sinais de informação e o mesmo número de sinais de perigo. Para este efeito, foi efetuado também o *plot* das *labels*, em formato de histograma, utilizando a biblioteca *seaborn*.

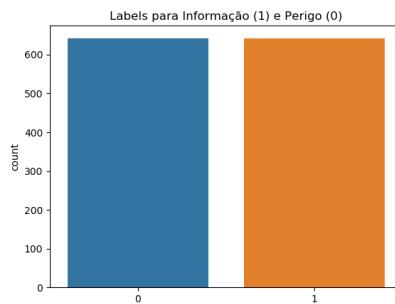


Figura 3.14: Visualização do número de classes.

Após certificarmos-nos que temos, efetivamente, duas classes com 642 sinais cada, observamos também a *shape* dos nossos dados, tendo sido obtidos os resultados apresentados na figura 3.15.

```
Shape of train images is: (1284, 150, 150, 3)
Shape of labels is: (1284,)
```

Figura 3.15: *Shape* de X e y.

Podemos concluir que o conjunto de treino é composto por um *array* de 4 dimensões - 1284 x 150 x 150 x 3 - o correspondente ao *batch\_size* (tamanho do conjunto), altura, largura e canais das imagens, respetivamente. A *shape* do *array* de imagens é importante para o modelo em *Keras* que irá ser construído, pois este recebe como *input* um vetor com altura, largura e canais.

### 3.8 Divisão do conjunto de treino e validação

Uma das coisas mais importantes a fazer antes do inicio da construção do modelo e do respetivo treino, é dividir o restante *dataset* em conjunto de treino e validação. Para esta divisão foi utilizado o método *train\_test\_split* da biblioteca do *sklearn.model\_selection*, como podemos ver no excerto 3.4.

```
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size = 0.20, random_state=2)
```

Listing 3.4: Divisão em conjunto de treino e validação.

Neste caso, foi atribuído 80% dos dados ao conjunto de treino e 20% ao conjunto de validação.

Depois desta divisão, verificamos novamente a *shape* dos dados, estando o resultado destas apresentado na figura 3.16.

```
Shape of train imgs: (1027, 150, 150, 3)
Shape of validation imgs: (257, 150, 150, 3)
Shape of train lables (1027,)
Shape of validation imgs: (257,)
```

Figura 3.16: *Shape* de X e y após a divisão dos conjuntos de teste e validação.

Após nova limpeza de memória às variáveis X e y, guardamos em duas variáveis (`ntrain` e `nval`) o tamanho dos conjuntos de treino e validação, respectivamente, para posterior uso. Foi também definido um `batch_size` de 32.

### 3.9 Arquitetura do modelo CNN

Nesta secção iremos abordar a arquitetura do modelo implementado, que se refere à forma como os *layers* convolucionais estão inseridos. Foi utilizada uma arquitetura popular, eficaz e simples denominada de *VGGnet*. Esta arquitetura foi proposta pelo *Visual Geometry Group*, grupo da Universidade de Oxford, em 2014.

O modelo em questão foi criado a partir da biblioteca do **Keras**. Esta é uma biblioteca de redes neurais de código aberto escrita em Python. É capaz de executar sobre o *TensorFlow*, o *Microsoft Cognitive Toolkit* ou o *Theano*. O **Keras** foi projetado para permitir experimentação rápida com redes neurais profundas, concentrando-se assim na fácil, modular e extensível utilização.

Foi utilizada uma pequena VGGnet, mas podemos notar que o tamanho do filtro vai aumentando à medida que as camadas aumentam. A arquitetura utilizada resume-se em: 32 → 64 → 128 → 512 - e a camada final é 1. Esta encontra-se apresentada no excerto 3.5.

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3,3), activation='relu', input_shape=(150,150,3)))
model.add(layers.MaxPooling2D((2,2)))
model.add(layers.Conv2D(64, (3,3), activation='relu'))
model.add(layers.MaxPooling2D((2,2)))
model.add(layers.Conv2D(128, (3,3), activation='relu'))
model.add(layers.MaxPooling2D((2,2)))
model.add(layers.Conv2D(128, (3,3), activation='relu'))
model.add(layers.MaxPooling2D((2,2)))
model.add(layers.Flatten())
model.add(layers.Dropout(0.5))
```

```

model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

```

Listing 3.5: Arquitetura do modelo CNN.

Inicialmente é criado um modelo de *layers* sequenciais. Criamos então um primeiro *layer* ao chamar a função `.add()` no modelo criado, passando então o tipo de *layer* que queremos, neste caso, um Conv2D *layer*. Este primeiro *layer* é chamado de *input layer* e conta com alguns parâmetros importantes, tais como:

- ***filters***: este é o tamanho da dimensão de saída, ou seja, o número de filtros de saída na convolução. Neste caso começamos por 32;
- ***kernel\_size***: especifica a altura e a largura da janela de convolução 2D, neste caso, 3 x 3;
- ***activation***: refere-se à função de ativação a ser usada na rede neuronal. Neste caso utilizamos a *Rectified Linear Unit* (ReLU), que é a função de ativação mais comum usada atualmente. Esta função melhora as redes neurais na medida em que acelera o processo de treino. O cálculo do gradiente é muito simples (0 ou 1, dependendo do sinal de x). Além disso, os passos computacionais da função ReLU são simples: qualquer elemento negativo é definido como 0,0 - sem exponenciais e sem operações de multiplicação ou divisão [5];
- ***input\_shape***: refere-se às dimensões com que redimensionamos as nossas imagens e o canal, neste caso é [150,150,3].

De seguida temos o *layer* MaxPooling2D. A função desta camada, como já abordado no capítulo 2, é reduzir o tamanho espacial dos recursos recebidos e, portanto, reduzir o número de parâmetros e a computação no modelo. Assim, este *layer* ajuda a reduzir adaptação excessiva (*overfitting*). O *overfitting* acontece, quando o nosso modelo memoriza os dados de treino. Deste modo, o modelo teria um excelente desempenho no tempo de treino, mas iria falhar no tempo de teste.

De seguida, foi acrescentada mais uma camada convolucional, desta vez com 64 filtros e feito o respetivo MaxPooling2D. O mesmo sucedeu-se para as restantes camadas, ambas com 128 filtros.

Após isso, foi declarado o **Flatten layer**. As camadas Conv2D extraem e aprendem os recursos espaciais, que são passados para uma camada densa depois de terem sido "*flattened*". Basicamente, esta camada transforma uma matriz bidimensional de recursos num vetor que pode ser alimentado num classificador de rede neuronal totalmente conectado, como podemos observar na figura 3.17.

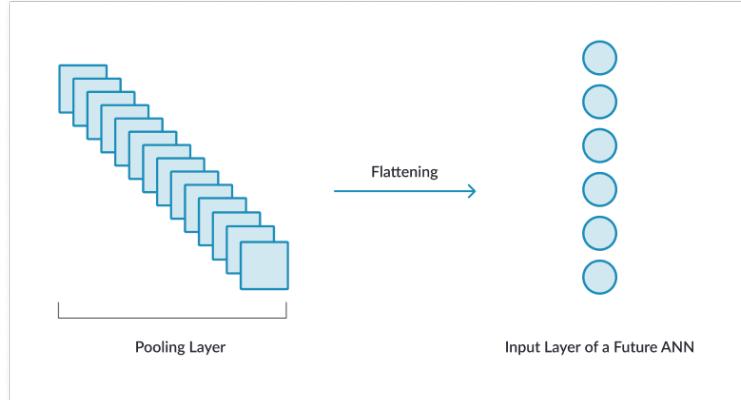


Figura 3.17: Camada *Flatten*.

De seguida, adicionamos uma camada *Dropout* com valor de 0,5. Esta camada descarta aleatoriamente algumas camadas nas redes neurais e aprende com a rede reduzida. Desta forma, a rede aprende a ser independente e não confiável numa única camada. Esta camada também auxilia no *overfitting*. O 0,5 significa descartar, aleatoriamente, metade das camadas.

De seguida foi adicionada uma camada densa, ou seja, totalmente conectada com 512 *inputs*.

O último *layer*, também ele denso, possui um tamanho de saída de 1 e uma função de ativação chamada *sigmoid*. Isto acontece porque estamos a tentar detetar se uma imagem é um sinal de informação ou de perigo, ou seja, queremos que o modelo mostre uma probabilidade de quão certa uma imagem é um sinal de informação e não de perigo. Para este efeito, pretendemos um valor probabilístico, em que valores mais altos significam que o classificador acredita que a imagem é um sinal de informação e valores mais baixos de perigo. A função *sigmoid* é ideal para esta tarefa, pois recebe um conjunto de números e retorna uma distribuição de probabilidades no intervalo de 0 a 1.

Para visualizarmos a arquitetura e o tamanho dos parâmetros da CNN implementada podemos utilizar o método `summary()` do Keras no objeto do modelo.

Na figura 3.18, podemos observar o número de parâmetros que pretendemos treinar, neste caso mais do que 3 milhões, e a arquitetura geral dos diferentes *layers*.

De seguida, foi necessário compilar o modelo com o método `compile()` do Keras, como podemos observar no excerto 3.6.

```
model.compile(loss='binary_crossentropy', optimizer = optimizers.  
RMSprop(lr=1e-4), metrics= ['acc'])
```

Listing 3.6: Compilação do modelo.

Neste método são passados três parâmetros:

- ***loss***: especificamos uma função de perda que o nosso otimizador irá minimizar. Neste caso, como estamos a trabalhar com um modelo apenas com

Model: "sequential_1"		
Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 148, 148, 32)	896
max_pooling2d_1 (MaxPooling2D)	(None, 74, 74, 32)	0
conv2d_2 (Conv2D)	(None, 72, 72, 64)	18496
max_pooling2d_2 (MaxPooling2D)	(None, 36, 36, 64)	0
conv2d_3 (Conv2D)	(None, 34, 34, 128)	73856
max_pooling2d_3 (MaxPooling2D)	(None, 17, 17, 128)	0
conv2d_4 (Conv2D)	(None, 15, 15, 128)	147584
max_pooling2d_4 (MaxPooling2D)	(None, 7, 7, 128)	0
flatten_1 (Flatten)	(None, 6272)	0
dropout_1 (Dropout)	(None, 6272)	0
dense_1 (Dense)	(None, 512)	3211776
dense_2 (Dense)	(None, 1)	513
<hr/>		
Total params: 3,453,121		
Trainable params: 3,453,121		
Non-trainable params: 0		
<hr/>		
None		

Figura 3.18: Sumário do modelo.

duas classes, usamos a perda de entropia cruzada binária (`binary_crossentropy`);

- **optimizer:** neste caso utilizamos o otimizador RMSProp com uma *learning rate* de 0.0001. Este otimizador faz parte de um processo chamado ajuste de hiperparâmetros (*hyperparameter tuning*), que pode ser a diferença entre um modelo real e um mais ingênuo. Hiperparâmetros são todos os parâmetros de um modelo que não são atualizados durante a aprendizagem e são usados para configurar o modelo (por exemplo, tamanho do espaço de *hash*, número de árvores de decisão e a sua profundidade, número de camadas de uma rede neuronal profunda, etc.) ou o algoritmo usado para diminuir a função de custo (taxa de aprendizagem para um algoritmo, etc.). Assim, o *hyperparameter tuning* consiste no problema de escolher automaticamente um conjunto de hiperparâmetros ideais para um algoritmo de aprendizagem;
- **metrics:** aqui é onde especificamos qual é a métrica que queremos usar para medir o desempenho do nosso modelo após o treino. Como estamos a lidar com um problema de classificação, a métrica de precisão (*acc*) é uma boa opção. Esta métrica é utilizada para medir o desempenho do modelo, sendo que esta depende do tipo de problema com o qual estamos a lidar [2].

## 3.10 Normalização

Finalmente, e antes de começarmos a treinar o modelo, precisamos de executar alguma normalização, como por exemplo, dimensionar os nossos valores de pixeis

das imagens para obtermos um desvio padrão unitário e uma média de 0. Para este efeito, foi utilizado o módulo `ImageDataGenerator` do Keras, que executa algumas funções importantes ao inserir imagens no modelo durante o treino.

Segundo o criador do Keras, François Chollet, o método `ImageDataGenerator` desta biblioteca, permite-nos configurar rapidamente geradores que transformam automaticamente ficheiros de imagens em *tensors* pré-processados, que podem ser embutidos diretamente nos modelos durante a fase de treino. Este método inclui diversas funcionalidades, tais como:

- Descodificar o conteúdo .jpg em grelhas de pixeis RGB;
- Converter estas grelhas em *tensors* de ponto flutuante;
- Dimensionar os valores dos pixeis (entre 0 e 255) para o intervalo [0,1] (as redes neuronais apresentam melhor desempenho com dados normalizados);
- Ajuda a aumentar facilmente as imagens (*data augmentation*). Isto é um recurso importante que irá ser utilizado, pois o nosso conjunto de dados de treino é pequeno.

Para este efeito foram criados dois geradores, um para o conjunto de treino e outro para o conjunto de validação, como podemos ver no excerto 3.7.

```
train_datagen = ImageDataGenerator(rescale=1./255, rotation_range=40, width_shift_range=0.2, height_shift_range=0.2, shear_range=0.2, zoom_range=0.2, horizontal_flip=True, )
val_datagen = ImageDataGenerator(rescale=1./255)
```

Listing 3.7: Normalização das imagens.

Como podemos observar, inicialmente passamos a opção de redimensionamento (*rescale*) para o objeto `ImageDataGenerator`. A opção *rescale* = 1./255 é um parâmetro muito importante, pois normaliza os valores do pixel da imagem inserindo-os no intervalo [0,1]. Isto ajuda o modelo a aprender e a atualizar os seus parâmetros gerais de forma mais eficiente. Os parâmetros seguintes são relativos à *data augmentation*. Estes indicam ao `ImageDataGenerator` para aplicar aleatoriamente alguma transformação na imagem. Isto irá ajudar a aumentar o nosso conjunto de dados e melhorar a generalização, prevenindo também assim o *overfitting*.

Na variável `val_datagen` também criamos um objeto `ImageDataGenerator` para o nosso conjunto de validação no entanto, neste caso, não realizamos *data augmentation*, realizando assim apenas a operação de redimensionamento.

Após termos os objetos `ImageDataGenerator` inicializados, criamos efetivamente os geradores a partir destes, passando os nossos conjuntos de treino e validação, como podemos ver no excerto 3.8.

```
train_generator = train_datagen.flow(X_train, y_train, batch_size=batch_size)
val_generator = val_datagen.flow(X_val, y_val, batch_size=batch_size)
```

Listing 3.8: Criação dos geradores.

Como é possível observar, foi chamado o método `flow()` nos geradores de dados anteriormente criados, passando os conjuntos X e y: X\_train e y\_train para o treino e X\_val e y\_val para a validação. O tamanho do *batch* informa o gerador de dados para levar apenas o tamanho especificado (32 no nosso caso) de imagens, de cada vez.

### 3.11 Treino do modelo

O treino do modelo foi feito evocando o método `fit()` do Keras, e passando alguns parâmetros, como podemos observar no excerto 3.9.

```
history = model.fit_generator(train_generator,
                               steps_per_epoch=ntrain // batch_size,
                               epochs=100,
                               validation_data=val_generator,
                               validation_steps=nval // batch_size)
```

Listing 3.9: Treino do modelo.

O primeiro parâmetro é referente ao objeto `ImageDataGenerator` do conjunto de treino, sendo este denominado de `train_generator`;

No segundo parâmetro (`steps_per_epoch`) é especificado o número de passos por iteração. Isto indica ao modelo quantas imagens queremos processar antes de fazer-mos uma atualização do gradiente à nossa função de custo. Neste caso, um total de 1027 imagens divididas pelo tamanho do *batch* definido a 32, irá dar aproximadamente 10 passos. Isto significa que faremos um total de 10 atualizações de gradiente no modelo, por iteração.

No terceiro parâmetro é definido o número de *epochs*, ou seja, de iterações. Uma iteração é um ciclo completo por todo o conjunto de treino. No nosso caso, uma *epoch* é alcançada quando fazemos 10 atualizações de gradiente, conforme especificado pelo parâmetro `steps_per_epoch`. Neste caso foram definidas 100 iterações.

De seguida, nos dois últimos parâmetros, indicamos o conjunto de validação e o tamanho dos passos neste dados, que neste caso é o tamanho do vetor `X_val`.

```

Epoch 83/100
[=====] - 1ls 350ms/step - loss: 0.2863 - acc: 0.8857 - val_loss: 0.4536 - val_acc: 0.8711
Epoch 84/100
[=====] - 1ls 340ms/step - loss: 0.2527 - acc: 0.8954 - val_loss: 0.4376 - val_acc: 0.8889
Epoch 85/100
[=====] - 1ls 340ms/step - loss: 0.2999 - acc: 0.8693 - val_loss: 0.1651 - val_acc: 0.8756
Epoch 86/100
[=====] - 1ls 350ms/step - loss: 0.2789 - acc: 0.8884 - val_loss: 0.2394 - val_acc: 0.9333
Epoch 87/100
[=====] - 1ls 340ms/step - loss: 0.2797 - acc: 0.8764 - val_loss: 0.2116 - val_acc: 0.9111
Epoch 88/100
[=====] - 1ls 360ms/step - loss: 0.3107 - acc: 0.8644 - val_loss: 0.3020 - val_acc: 0.9044
Epoch 89/100
[=====] - 1ls 350ms/step - loss: 0.2765 - acc: 0.8854 - val_loss: 0.2520 - val_acc: 0.9208
Epoch 90/100
[=====] - 1ls 350ms/step - loss: 0.2986 - acc: 0.8673 - val_loss: 0.9710 - val_acc: 0.9156
Epoch 91/100
[=====] - 1ls 350ms/step - loss: 0.2715 - acc: 0.8874 - val_loss: 0.1156 - val_acc: 0.9180
Epoch 92/100
[=====] - 1ls 350ms/step - loss: 0.2739 - acc: 0.8874 - val_loss: 0.1782 - val_acc: 0.9244
Epoch 93/100
[=====] - 1ls 350ms/step - loss: 0.2749 - acc: 0.8884 - val_loss: 0.2166 - val_acc: 0.9156
Epoch 94/100
[=====] - 1ls 350ms/step - loss: 0.3065 - acc: 0.8442 - val_loss: 0.7004 - val_acc: 0.6652
Epoch 95/100
[=====] - 1ls 330ms/step - loss: 0.5946 - acc: 0.7474 - val_loss: 0.8955 - val_acc: 0.8844
Epoch 96/100
[=====] - 1ls 350ms/step - loss: 0.2895 - acc: 0.8880 - val_loss: 0.4275 - val_acc: 0.6133
Epoch 97/100
[=====] - 1ls 350ms/step - loss: 0.2954 - acc: 0.8844 - val_loss: 0.3882 - val_acc: 0.9067
Epoch 98/100
[=====] - 1ls 340ms/step - loss: 0.2579 - acc: 0.8965 - val_loss: 0.5094 - val_acc: 0.9111
Epoch 99/100
[=====] - 1ls 340ms/step - loss: 0.2777 - acc: 0.8874 - val_loss: 0.1680 - val_acc: 0.6022
Epoch 100/100
[=====] - 1ls 350ms/step - loss: 0.2849 - acc: 0.8854 - val_loss: 0.0891 - val_acc: 0.8984
Tempo de Treino: 19.054805890666667 min

```

Figura 3.19: Treino do modelo no terminal.

Na figura 3.19, conseguimos observar como aparece o processo de treino impresso no terminal do IDE utilizado.

Após o treino estar concluído, são guardados os pesos do modelo através do método `save_weights()` do Keras e o modelo em si, através do método `save()` da mesma biblioteca.

Após treinar um modelo utilizando o Keras, são sempre calculadas e guardadas as métricas que foram especificadas quando compilamos o modelo, numa variável chamada *history*. É possível extraíremos estes valores, mostrando-os graficamente através da biblioteca do `matplotlib`. Este objeto `history` contém todas as atualizações que aconteceram durante o processo de treino.

Estes valores foram extraídos conforme mostra o excerto 3.10.

```
acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(acc) + 1) #tamanho da epoch a partir do
                               numero de valores na lista "acc"
```

Listing 3.10: Extração das métricas no treino do modelo.

## 3.12 Teste do modelo

Por último, foi efetuado o teste ao modelo com as imagens presentes no conjunto de teste.

Para este efeito, foi feito o mesmo pré-processamento que foi aplicado nos conjuntos de treino e validação, como podemos observar no excerto 3.11.

```
X_test, y_test = read_and_process(test_imgs[49:64])
x = np.array(X_test)
test_datagen = ImageDataGenerator(rescale=1./255)
```

Listing 3.11: Pré-processamento no conjunto de teste.

Inicialmente lemos e convertemos o conjunto de imagens entre a posição 49 e 64 do conjunto de teste numa lista de *arrays* através da função `read_and_process()` previamente definida. Neste caso, o `y_test` vai estar vazio porque o conjunto de teste não tem *labels*. De seguida convertemos essa lista num *array numpy*, e por último, inicializamos um objeto `ImageDataGenerator` a partir desse conjunto de teste, aplicando apenas a operação de normalização dos pixels da imagem.

Após este pré-processamento, criamos um *loop* através do ciclo `for`, que vai iterar as imagens do gerador e, com estas, fazer as previsões e dar *display* aos resultados. Podemos observar este processo no excerto 3.12.

```
i=0
text_labels = [] #vai conter os labels que vao ser previstos
plt.figure(figsize=(30,20))

for batch in test_datagen.flow(x, batch_size=1):
    pred = model.predict(batch)
    if pred > 0.5:
```

```

    text_labels.append('sinal de informacao')
else:
    text_labels.append('sinal de perigo')
plt.subplot(5 / columns + 2, columns, i+1)
plt.title('Isto é um ' + text_labels[i])
imgplot = plt.imshow(batch[0])
i+=1
if i % 15 == 0:
    break
plt.show()

```

Listing 3.12: Previsões e *display* das imagens de teste.

A previsão é feita em cada imagem do conjunto de teste passada pelo objeto `ImageDataGenerator` através do método `predict()` do Keras. A variável `pred` é uma probabilidade do quão certo o modelo está de que a imagem atual é um sinal de perigo ou informação. Como atribuímos às *labels* de informação um valor de 1, ou seja, uma alta probabilidade (pelo menos maior do que 0,5), isto significa que o modelo está bastante confiante de que a imagem seria, neste caso, um sinal de informação, contrário interpreta-a como um sinal de perigo.

Assim, para este efeito, foi criada uma instrução `if-else` que anexa a string 'sinal de perigo' se a probabilidade for maior que 0,5, caso contrário, anexa 'sinal de informação' à variável `text_label`. Este processo está a ser feito para conseguirmos adicionar um título à imagem quando fazemos *display* da mesma através da biblioteca `matplotlib`.

## Capítulo 4

# Resultados obtidos e Testes experimentais

Após a implementação do projeto referido no capítulo 3, e do treino do modelo estar concluído, foram obtidos diversos resultados graficamente através da biblioteca `matplotlib.pyplot`. Foram também efetuados alguns testes, variando os fatores do número de *epochs*, do *batch* e do otimizador.

### 4.1 Resultados obtidos

O projeto foi efetuado, inicialmente, com um *batch\_size* de 32, com 100 iterações, utilizando o algoritmo RMSProp e o conjunto entre a imagem 50 à 65.

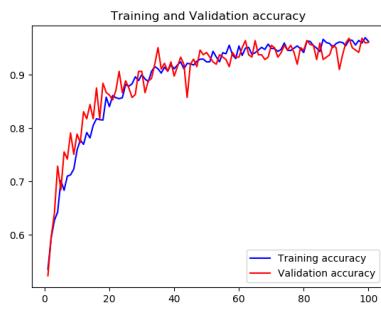


Figura 4.1: *Accuracy* do treino e validação.

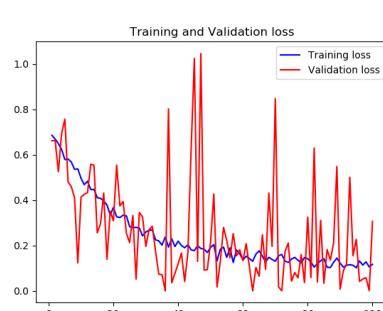


Figura 4.2: Custo do treino e validação.

A primeira coisa que é possível observar a partir dos gráficos presentes nas figuras 4.1 e 4.2, é que não está a acontecer *overfitting*, pois a precisão (*accuracy*)

do treino e da validação estão relativamente próximas. É também de notar que a precisão aumenta continuamente à medida que as iterações aumentam, dando-nos a entender que se aumentar-mos o número de iterações, provavelmente irá dar uma precisão ainda mais alta. Relativamente ao gráfico de custo, este também não está a fazer *overfitting* pois, por norma, este valor vai diminuindo, havendo no entanto mais oscilações. O custo irá, provavelmente, diminuir se aumentarmos o número de iterações.

Este teste foi efetuado para o conjunto entre a primeira imagem e a décima-quinta, estando os resultados presentes na figura 4.3.



Figura 4.3: Figuras 50 à 65 - resultados obtidos.

Neste caso, e fazendo uma avaliação empírica das imagens obtidas e correspondentes *labels* previstas, conseguimos observar que o modelo acertou em todas as 15 figuras testadas, atribuindo a classe correta (perigo ou informação) a que cada uma pertence.

## 4.2 Testes experimentais

### 4.2.1 Menor número de iterações

Nesta sub-secção iremos demonstrar os resultados obtidos, com os mesmos parâmetros que na secção 4.1, à exceção do número de iterações que foi diminuído para apenas 20.

Nos gráficos apresentados nas figuras 4.4 e 4.5, é possível constatar que existe muito menos precisão, não havendo portanto uma linearidade tão grande na função da validação relativamente à do treino. O mesmo acontece para o gráfico do custo, que tem mais oscilações.

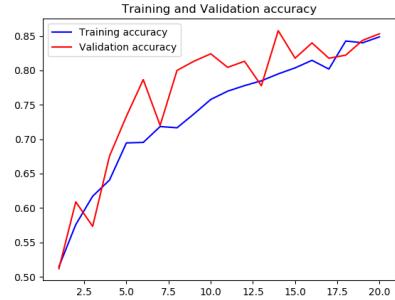


Figura 4.4: *Accuracy* do treino e validação - 20 iterações.

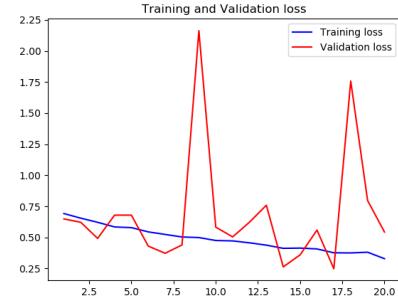


Figura 4.5: Custo do treino e validação - 20 iterações.



Figura 4.6: Imagem 50 à 65 - 20 iterações.

Neste caso, e fazendo uma avaliação empírica dos resultados obtidos, conseguimos notar que o modelo errou 2 imagens de 15 para o conjunto de imagens em questão, concluindo assim que este precisa de mais iterações para aprender de forma mais precisa.

#### 4.2.2 Maior número de iterações

Nesta sub-secção iremos demonstrar os resultados obtidos, com os mesmos parâmetros que na secção 4.1, à exceção do número de iterações que foi aumentado para 200. O conjunto de fotos testadas foi também alterado para entre as posições 150 e 165, visto que o modelo acertou todos os sinais aquando das 100 iterações testadas.

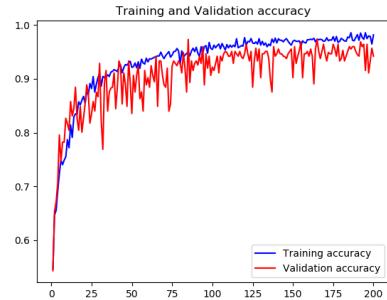


Figura 4.7: *Accuracy* do treino e validação - 200 iterações.

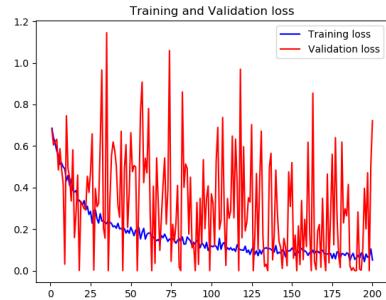


Figura 4.8: Custo do treino e validação - 200 iterações.

Nos gráficos apresentados nas figuras 4.7 e 4.8, é possível constatar que existe uma grande precisão do conjunto de validação comparativamente ao conjunto de teste não havendo, portanto, *overfitting*. No gráfico que representa o valor de custo por iteração, vemos que o conjunto de validação tem bastantes oscilações relativamente ao de treino começando ambos, no entanto, numa posição de custo idêntica e acabando também da mesma forma.

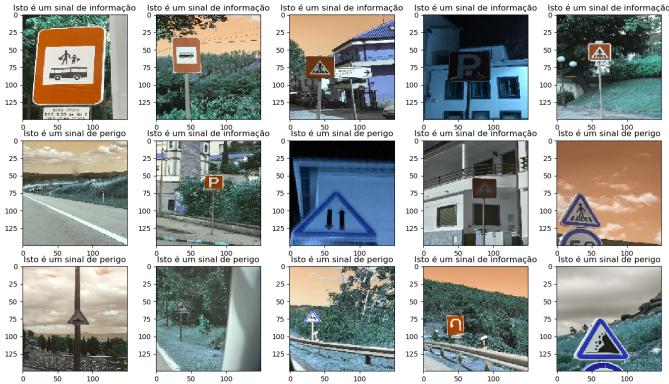


Figura 4.9: Imagem 150 à 165 - 200 iterações.

De acordo com os resultados apresentados na figura 4.9, e fazendo também uma avaliação empírica dos resultados obtidos, conseguimos observar que o modelo acertou em todas as 15 figuras testadas, atribuindo a classe correta (perigo ou informação) a que cada uma pertence.

### 4.2.3 Menor batch size

Nesta sub-secção iremos demonstrar os resultados obtidos, com os mesmos parâmetros que na secção 4.1, à exceção do tamanho do *batch* que foi diminuido para 10. O conjunto de fotos testadas foi também alterado para entre as posições 304 e 319.

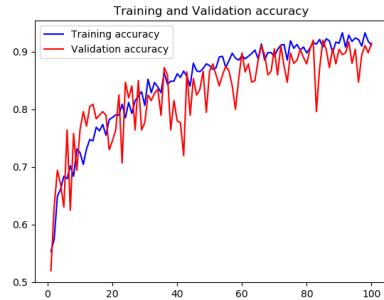


Figura 4.10: *Accuracy* do treino e validação - *batch size* de 10.

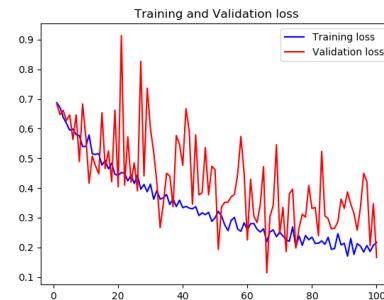


Figura 4.11: Custo do treino e validação - *batch size* de 10.

Nos gráficos apresentados nas figuras 4.10 e 4.11, é possível constatar que obtivemos, em geral, melhores resultados, relativamente ao resultados inicialmente obtidos na secção 4.1. Não existe *overfitting* em nenhum dos gráficos, sendo que os conjuntos de validação e treino andam com valores relativamente próximos.



Figura 4.12: Imagem 304 à 319 - *batch size* de 10.

De acordo com os resultados apresentados na figura 4.12, e fazendo também uma avaliação empírica dos resultados obtidos para o conjunto de imagens em questão, conseguimos observar que o modelo errou 1 imagem de 15.

#### 4.2.4 Maior *batch size*

Nesta sub-secção iremos demonstrar os resultados obtidos, com os mesmos parâmetros que na secção 4.1, à exceção do tamanho do *batch* que foi aumentado para 100. Paralelamente à sub-secção anterior, o conjunto de fotos testadas foi também alterado para entre as posições 304 e 319.

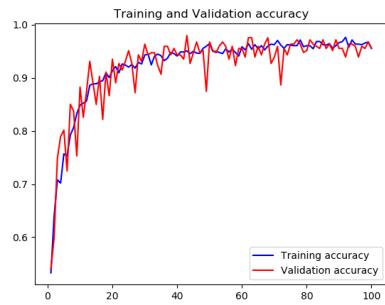


Figura 4.13: *Accuracy* do treino e validação - *batch size* de 100.

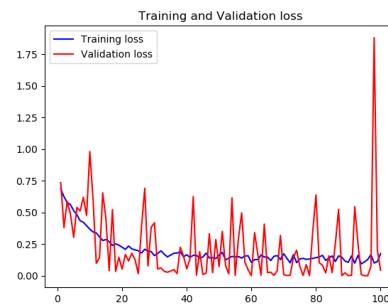


Figura 4.14: Custo do treino e validação - *batch size* de 100.

Nos gráficos apresentados nas figuras 4.13 e 4.14, é possível constatar que obtivemos melhores resultados relativamente aos gráficos referentes a um *batch size* de 10, havendo em geral uma grande semelhança entre os valores obtidos nos conjuntos de treino e validação, não existindo assim *overfitting*.



Figura 4.15: Imagem 304 à 319 - *batch size* de 100.

De acordo com os resultados apresentados na figura 4.15, e fazendo também uma avaliação empírica dos resultados obtidos para o conjunto de imagens em questão, conseguimos observar que o modelo errou 1 imagem de 15, tendo errado a mesma imagem que errou com um tamanho de *batch* de 10.

#### 4.2.5 Otimizador Adam

Nesta sub-secção iremos demonstrar os resultados obtidos, com os mesmos parâmetros que na secção 4.1, à exceção otimizador que foi mudado para o Adam. O conjunto de fotos testadas foi também alterado para entre as posições 185 e 200.

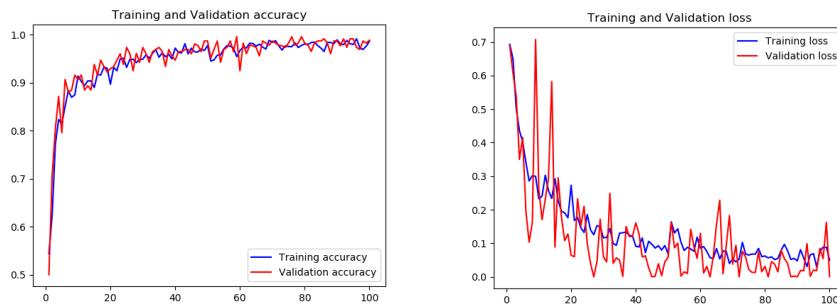


Figura 4.16: *Accuracy* do treino e validação - otimizador Adam.

Figura 4.17: Custo do treino e validação - otimizador Adam.

Nos gráficos apresentados nas figuras 4.16 e 4.17, é possível constatar que

obtivemos melhores resultados relativamente ao resultados inicialmente obtidos na secção 4.1, sendo que os conjuntos de treino e validação apresentam valores bastante próximos, principalmente no gráfico que apresenta o valor da *accuracy* por iteração. Na observação dos gráficos em cima referidos, é possível concluir que o otimizador Adam é geralmente mais estável e aprende mais rapidamente, relativamente ao otimizador inicialmente utilizado, RMSProp.

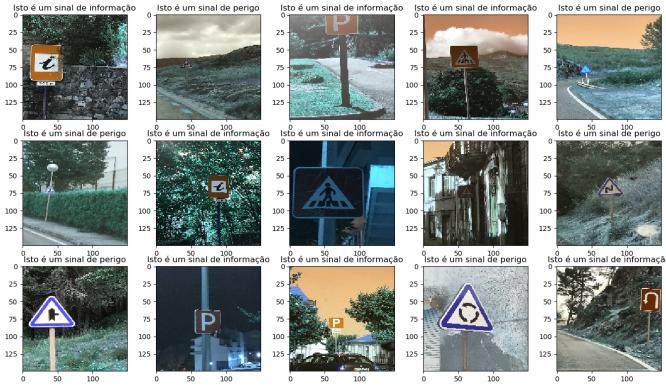


Figura 4.18: Imagem 185 à 200 - otimizador Adam.

De acordo com os resultados apresentados na figura 4.18, e fazendo também uma avaliação empírica dos resultados obtidos, conseguimos observar que o modelo acertou em todas as 15 figuras testadas, atribuindo a classe correta (perigo ou informação) a que cada uma pertence.

#### 4.2.6 Otimizador SGD

Nesta sub-secção iremos demonstrar os resultados obtidos, com os mesmos parâmetros que na secção 4.1, à exceção otimizador que foi mudado para o SGD. O conjunto de fotos testadas foi também alterado para entre as posições 215 e 230.

Nos gráficos apresentados nas figuras 4.19 e 4.20, é possível constatar que obtivemos ligeiramente piores resultados no gráfico de *accuracy* relativamente ao resultados inicialmente obtidos na secção 4.1 e na sub-secção anterior, sendo que os conjuntos de treino e validação apresentam, ainda assim, valores relativamente próximos. Na observação dos gráficos em cima referidos, é possível concluir que o otimizador SGD é geralmente menos estável e aprende mais lentamente, relativamente ao otimizador inicialmente utilizado, RMSProp, e ao otimizador testado na sub-secção anterior - Adam. No entanto, e ainda comparativamente ao otimizador RMSProp, o gráfico das funções de custo com o SGD apresenta menos oscilações entre ambos os conjuntos de treino e validação,

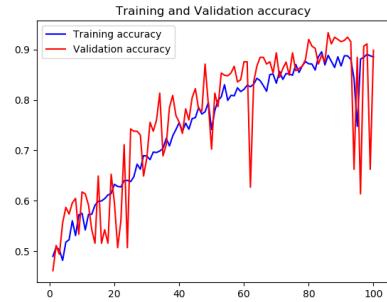


Figura 4.19: *Accuracy* do treino e validação - otimizador SGD.

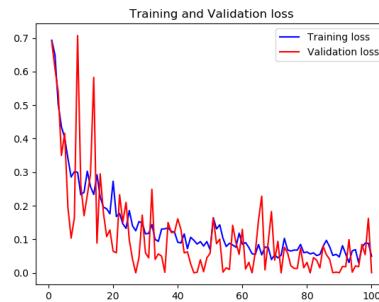


Figura 4.20: Custo do treino e validação - otimizador SGD.

tendo uma queda do custo bastante mais acentuada. Nenhum dos gráficos está a fazer *overfitting*.

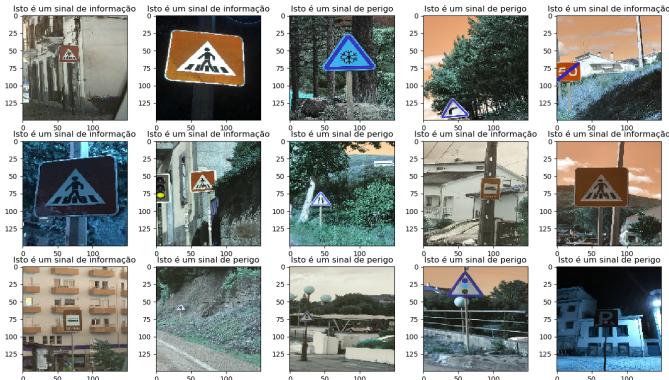


Figura 4.21: Imagem 215 à 230 - otimizador SGD.

De acordo com os resultados apresentados na figura 4.21, e fazendo também uma avaliação empírica dos resultados obtidos para o conjunto de imagens em questão, conseguimos observar que o modelo errou 1 imagem de 15.

## Capítulo 5

# Conclusão e trabalho futuro

A IA tem testemunhado um crescimento gigante entre as capacidades de Humanos e máquinas. Um dos principais objetivos no domínio da Visão Computacional, é permitir que as máquinas percessem o Mundo de maneira o mais similar possível aos Seres Humanos, utilizando esse conhecimento para diversas tarefas, tais como sistemas de reconhecimento, análise e classificação de imagens e vídeos, processamento de linguagem natural, etc. Os avanços neste domínio, com auxílio do *Deep Learning*, foram construídos e aperfeiçoados ao longo do tempo, principalmente através de um algoritmo específico - uma CNN.

Neste relatório foi dada, inicialmente, uma introdução ao funcionamento destas CNN. Conseguimos assim entender que uma CNN recebe um determinado *input* (geralmente uma imagem), passa-o numa série de filtros e camadas, procurando assim obter um *output* que possa ser classificado.

Este trabalho foi bastante importante para aprender e entender com muito mais detalhe o modo de funcionamento de uma CNN nos seus vários processos de otimização, desde a importância do pré-processamento das imagens, até à obtenção final do modelo final a ser classificado.

É importante frisar o impacto positivo que a utilização da GPU, comparativamente ao CPU, teve no tempo de treino do modelo. Se a GPU utilizada fosse mais potente, era possível treinar o modelo em questão de poucos minutos, bem como adicionar um *dataset* maior sem perder desempenho ao nível de tempo de execução.

Foram também mostrados os resultados obtidos e efetuados diversos testes que variaram parâmetros como o número de iterações, *batch size* e otimizador utilizado. Destes, chegamos à conclusão que o melhor otimizador, para o problema em questão, é o Adam por apresentar uma velocidade de aprendizagem superior, para um mesmo valor de taxa de aprendizagem, relativamente aos restantes. Chegámos também à conclusão que o ideal é ter, no mínimo, 100 iterações, de forma ao modelo ser mais o mais preciso possível. Quanto ao tamanho do *batch size*, obtivemos resultados mais ou menos semelhantes nos casos testados, sendo que os resultados para um *batch size* de 100 foram ligeiramente melhores do que os restantes.

Concluo assim, que todos os objetivos do presente trabalho foram concluídos com sucesso sendo que, como trabalho futuro, gostaria bastante de estender o modelo para mais classes de sinais, possivelmente até abrigar todas as existentes.

# Bibliografia

- [1] Alex Fernandes Mansano. O que é uma Rede Neural Convolucional?, 2017. [Online] <https://pt.linkedin.com/pulse/o-que-%C3%A9-um-rede-neural-convolucionar-alex-fernandes-mansano>. Último acesso a 2 de Julho de 2020.
- [2] Alois Bissuel. Hyper-parameter optimization algorithms: a short review, 2019. [Online] <https://medium.com/criteo-labs/hyper-parameter-optimization-algorithms-2fe447525903>. Último acesso a 3 de Julho de 2020.
- [3] Simon Haykin. *Neural Networks and Learning Machines*. ISBN: 978-0-13-147139-9. Pearson Education, Inc., 3 edition, 2009.
- [4] NVIDIA. CUDA Toolkit. [Online] <https://developer.nvidia.com/cuda-toolkit>. Último acesso a 2 de Julho de 2020.
- [5] Sycorax. Why do we use ReLU in neural networks and how do we use it?, 2018. [Online] <https://stats.stackexchange.com/questions/226923/why-do-we-use-relu-in-neural-networks-and-how-do-we-use-it>. Último acesso a 3 de Julho de 2020.