

# **Visão por Computador e Processamento de Imagen**

(1º ano de Mestrado em Computação Gráfica)

## **Trabalho 1**

Relatório

Alexandre Flores  
(PG50165)

Pedro Alves  
(A93272)

Rui Armada  
(PG50737)

7 de julho de 2023

## **Resumo**

O presente relatório detalha o desenvolvimento e aplicação de técnicas de *Deep Learning* para o desenvolvimento de modelos de classificação de imagens ilustrativas de diversos sinais de trânsito, bem como uma análise dos resultados por eles obtidos.

# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>2</b>
<b>2</b>	<b>Preparação dos modelos e tratamento de dados</b>	<b>3</b>
2.1	Estratégia geral . . . . .	3
2.2	Estrutura da rede e hiperparâmetros . . . . .	6
<b>3</b>	<b>Estratégias desenvolvidas</b>	<b>7</b>
3.1	GTSRB_V1: Notebook de controlo . . . . .	7
3.2	GTSRB_V2: Dynamic data augmentation . . . . .	7
3.3	GTSRB_V3: Massive data augmentation . . . . .	9
3.4	GTSRB_V4: Stacked Ensemble . . . . .	10
<b>4</b>	<b>Resultados obtidos</b>	<b>11</b>
4.1	Notebook 1 . . . . .	11
4.2	Notebook 2 . . . . .	12
4.3	Notebook 3 . . . . .	14
4.4	Notebook 4 . . . . .	15
4.5	Conclusões e análise . . . . .	15
<b>5</b>	<b>Conclusão Final</b>	<b>16</b>

# Capítulo 1

## Introdução

No âmbito do estudo da visão por computador, procuramos desenvolver modelos de classificação de imagens de sinais de trânsito. Estes modelos são treinados com fotografias rudimentares de sinais de trânsito, obtidas do *dataset GTSRB* referenciado no enunciado do projeto. Quando fornecidos com um dataset do mesmo tipo, o modelo deve ser capaz de classificar o sinal em 1 de 43 tipos diferentes de sinais com um grau de confiança alto.

O presente relatório irá abordar a estratégia adotada para o desenvolvimento destes modelos. Para este efeito, o próximo capítulo explica de um modo geral a estratégia adotada para cada um dos modelos, sendo que o terceiro capítulo demonstra e analisa os resultados obtidos.

# Capítulo 2

## Preparação dos modelos e tratamento de dados

### 2.1 Estratégia geral

Ao longo do desenvolvimento deste projeto, foram criados quatro *notebooks* diferentes correspondentes a estratégias diferentes. Os primeiros três *notebooks* que foram desenvolvidos consistem no treino de um modelo com diferentes tipos de *data augmentation*, enquanto o quarto consiste na preparação de um *ensemble* composto pelos outros três modelos. Todos os modelos individuais partilham os mesmos hiperparâmetros, escolhidos após uma breve comparação de resultados com base no trabalho realizado nas aulas e alguns testes realizados nos primeiros treinos feitos neste *dataset*.

Portanto, para conseguir tratar os dados foi preciso realizar uma conversão do formato das imagens. O *dataset* fornecido é composto por 43 pastas, que correspondem às classes do *dataset*, que contêm várias imagens no formato .ppm. Contudo, de forma a ser possível utilizar as técnicas lecionadas na disciplina, foi necessário converter os ficheiros para um formato que fosse suportado. Assim sendo, foi desenvolvida a seguinte função que converte todos os ficheiros .ppm para .png:

```
1 def convert_to_png(data_path, convert=True):
2     if convert:
3         for i in range (NUM_CLASSES):
4             image_path = os.path.join(data_path, format(i, '05d'))
5             files = os.listdir(image_path) #['00001_00000.ppm', ...]
6
7             for file in files:
8                 try:
9                     image = Image.open(os.path.join(image_path,file))
10                    image.save(f'{image_path}/{file.split('.')[0]}.png')
11                except:
12                    pass
13            print(f'Finished converting all files in {data_path} into png files.')
14        else:
15            print(f'Dataset: {data_path} remains unchanged.'
```

Listing 2.1: Função de conversão de formato

Posteriormente, foram carregadas os dados de treino e de teste da seguinte forma:

```
16 datasetV1 = keras.preprocessing.image_dataset_from_directory(  
17     f'{data_path}',  
18     batch_size=32,  
19     label_mode='categorical',  
20     image_size=DIMENTION,  
21     shuffle=True  
22 )  
23  
24 normalize = keras.layers.experimental.preprocessing.Rescaling(1.0/255)  
25  
26 datasetV1 = datasetV1.map(lambda x,y: (normalize(x), y))  
27  
# Existem celulas entre estes dois carregamentos  
28  
29 testset = tf.keras.preprocessing.image_dataset_from_directory(  
30     f'{test_path}',  
31     image_size=DIMENTION,  
32     batch_size=BATCH_SIZE,  
33     label_mode='categorical',  
34     shuffle=True  
35 )  
36  
37  
38 testset = testset.map(lambda x, y: (normalize(x), y))
```

Listing 2.2: Carregamento de dados

Antes de mais, é importante verificar se as imagens presentes no *dataset* se encontram devidamente carregadas. Para isto, é utilizada a função `show_batch`, utilizada durante as aulas, para observar o que foi carregado.

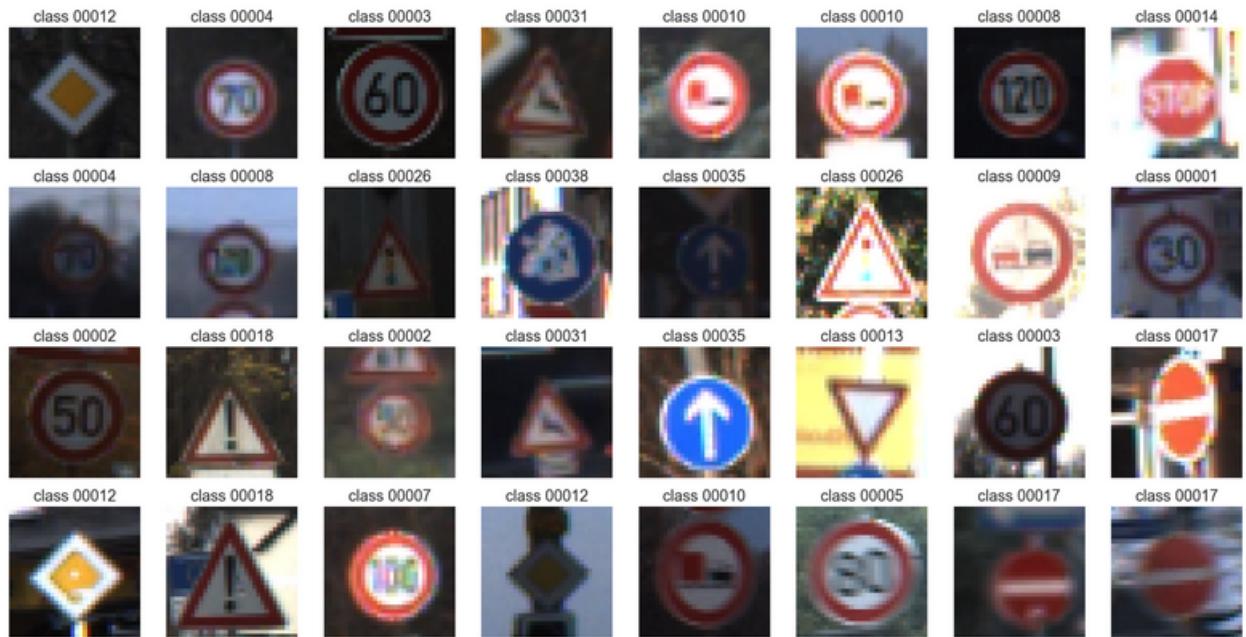


Figura 2.1: Imagens carregadas

Por último, foi considerada benéfica a criação de um conjunto de dados de validação que seriam usados como comparação com os dados de treino. Como não foram fornecidas imagens de validação foi realizado um `split` manual dos dados de treino.

- 80% Treino
- 20% Validação

```
39 train_size = int(0.8 * dataset_length/BATCH_SIZE)
40 val_size = int(0.2 * dataset_length/BATCH_SIZE)
41
42 train_dataset = datasetV1.take(train_size)
43 val_dataset = datasetV1.skip(train_size)
44
45 val_len = val_dataset.cardinality().numpy()
46 train_len = train_dataset.cardinality().numpy()
47
48 print(f'Train_Set length: {train_len}')
49 print(f'Valid_Set length: {val_len}')
```

Listing 2.3: Criação do validation set

Com isto, é necessário formatar os dados de validação de forma a conseguir um treino adequado dos dados.

```
50 def process_dataset(dataset , data_size):
51     data = dataset
52     data = data.cache()
53     data = data.shuffle(buffer_size = data_size)
54     data = data.prefetch(buffer_size=AUTOTUNE)
55
56     return data
57
58 # Existem celulas pelo meio
59
60 val_dataset = process_dataset(val_dataset , val_len)
```

Listing 2.4: Formatação dos dados de validação

## 2.2 Estrutura da rede e hiperparâmetros

Foram desenvolvidos vários modelos que obtiveram resultados diversos ao longo do desenvolvimento do projeto. Sendo assim, optámos por aplicar uma única estrutura de rede neuronal para todas as estratégias adotadas após algumas comparações de *performace* com hiperparâmetros e distribuições de *layers* distintas.

O modelo final desenvolvido é o seguinte:

```
61 def create_model():
62     modelLogits = Sequential()
63
64     modelLogits.add(Conv2D(128, KERNEL_SIZE, padding='same', input_shape=(32, 32, 3)))
65     modelLogits.add(LeakyReLU(alpha=0.01))
66     modelLogits.add(BatchNormalization())
67     modelLogits.add(Dropout(0.5))
68
69     modelLogits.add(Conv2D(196, KERNEL_SIZE))
70     modelLogits.add(LeakyReLU(alpha=0.01))
71     modelLogits.add(MaxPooling2D(pool_size=(2,2), padding='same'))
72     modelLogits.add(BatchNormalization())
73     modelLogits.add(Dropout(0.5))
74
75     modelLogits.add(Conv2D(256, KERNEL_SIZE))
76     modelLogits.add(LeakyReLU(alpha=0.01))
77     modelLogits.add(MaxPooling2D(pool_size=(2,2), padding='same'))
78     modelLogits.add(MaxPooling2D(pool_size=(2,2), padding='same'))
79     modelLogits.add(Dropout(0.5))
80
81     modelLogits.add(Flatten())
82     modelLogits.add(LeakyReLU(alpha=0.00))
83     modelLogits.add(Dense(384))
84     modelLogits.add(LeakyReLU(alpha=0.00))
85     modelLogits.add(Dropout(0.5))
86
87     modelLogits.add(Dense(43))
88
89     output = Activation('softmax')(modelLogits.output)
90
91     model = tf.keras.Model(modelLogits.inputs, output)
92
93     opt = Adam(learning_rate=0.0001)
94     model.compile(optimizer = opt, loss='categorical_crossentropy', metrics=['accuracy'])
95     return model, modelLogits
```

Listing 2.5: Estrutura de rede utilizada

Esta rede foi sempre treinada ao longo de 10 épocas, um valor que foi chegado após ser obeservado *overfitting* com totais de épocas maiores.

# Capítulo 3

## Estratégias desenvolvidas

### 3.1 GTSRB\_V1: Notebook de controlo

No primeiro *notebook*, o ”grupo de controlo”, os dados de *input* foram formatados de forma similar aos dados de validação, mas não foram aplicadas alterações às imagens em si.

```
96 train_dataset = train_dataset.cache()  
97 train_dataset = train_dataset.shuffle(buffer_size=train_len)  
98 train_dataset = train_dataset.prefetch(buffer_size=train_len)
```

Listing 3.1: Formatação dos dados de treino

Desta forma é possível comparar os resultados entre as outras estratégias de redes individuais, avaliando a eficácia da *data augmentation* efetuada e outras técnicas utilizadas.

### 3.2 GTSRB\_V2: Dynamic data augmentation

No segundo *notebook* foi explorada *dynamic data augmentation*. Aqui o objetivo é gerar diferentes versões das amostras originais de dados dentro da mesma época de treino. Esta abordagem serve para introduzir diversidade e variabilidade ao treino do modelo, evitando *overfitting* e melhorando a capacidade do modelo para generalizar para dados novos. É importante notar que os dados, que serão processados, são os dados de treino após o *split* de 80% e sem qualquer tipo de formatação aplicada aos mesmos. Neste caso, é aplicado um *shuffle* aos dados de treino e depois são aplicadas as seguintes transformações (com valores aleatórios) às imagens:

- Luminosidade
- Contraste
- Saturação
- Tom

Posteriormente, os dados, agora processados, são formatados de forma a poderem ser utilizados para o treino do modelo.

```

99 data_augmented = train_dataset
100
101 data_augmented = data_augmented.cache()
102 data_augmented = data_augmented.shuffle(buffer_size = dataset_length)
103 data_augmented = data_augmented.map(process_image)
104 data_augmented = data_augmented.prefetch(buffer_size=10200)
105
106 data_aug_len = data_augmented.cardinality().numpy()
107
108 print(f'Augmented Length: {data_aug_len}')

```

Listing 3.2: Dynamic Data Augmentation

Em que a função `process_image`, que aplica as transformações, é definida da seguinte maneira:

```

109 def process_image(image, label):
110     # change brightness
111     image = tf.clip_by_value(tfa.image.random_hsv_in_yiq(image, 0.0, 1.0, 1.0,
112                             0.1, 3.0), 0, 1)
113
114     # change contrast
115     image = tf.clip_by_value(tf.image.random_contrast(image, lower=0.1, upper
116                             =3.0, seed=None), 0, 1)
117
118     # change hue, saturation and value
119     image = tfa.image.random_hsv_in_yiq(image, 0.2, 0.4, 1.1, 0.4, 1.1)
120
121     image = tf.clip_by_value(image, 0, 1)
122
123     return image, label

```

Listing 3.3: Process image

Portanto, após a aplicação destes filtros, as imagens presentes nos dados de treino ficam da seguinte forma:

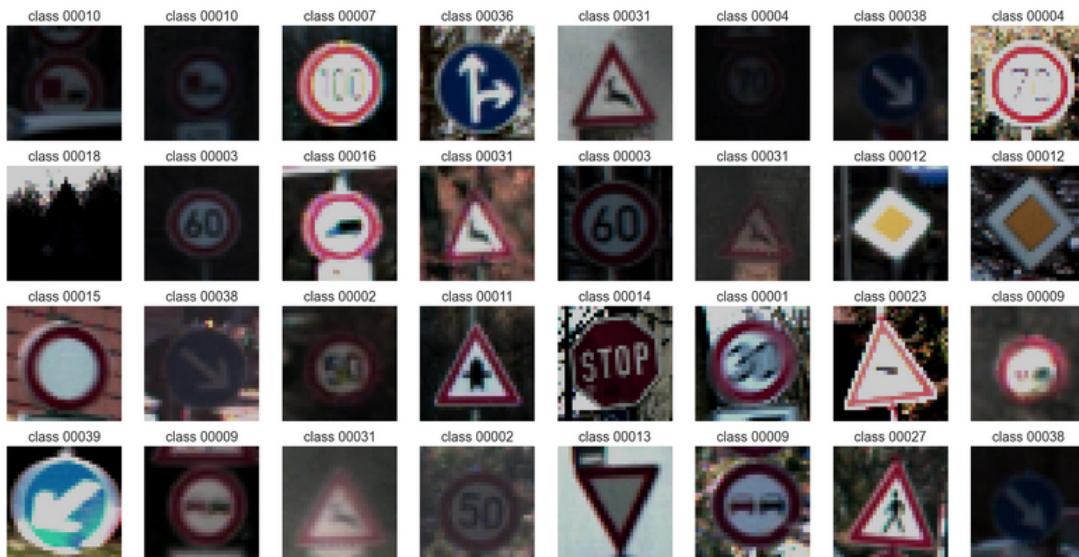


Figura 3.1: Imagens de treino após *dynamic data augmentation*

### 3.3 GTSRB\_V3: Massive data augmentation

No terceiro *notebook* é utilizada *massive data augmentation*. Aqui, a ideia é manter os dados originais no *dataset*, sendo que cada transformação é acrescentada ao *dataset* original, em vez de substituir todos os dados com as imagens transformadas. Portanto, são aplicadas múltiplas transformações ao conjunto de dados original que são concatenadas num único *dataset*. Em concreto, foram experimentados os seguintes tipos de transformações:

- Luminosidade
- Saturação
- Contraste
- Tom
- Rotação
- *Shear*
- Translações
- *Crop*

```
121 data_agumented = train_dataset
122
123 # color ops
124 data_agumented = data_agumented.map(process_brightness)
125 data_agumented = data_agumented.concatenate(train_dataset.map(process_contrast))
126 data_agumented = data_agumented.concatenate(train_dataset.map(process_hue))
127 data_agumented = data_agumented.concatenate(train_dataset.map(process_saturation))
128
129 #geometry ops
130 data_agumented = data_agumented.concatenate(train_dataset.map(process_rotate))
131 data_agumented = data_agumented.concatenate(train_dataset.map(process_shear))
132 data_agumented = data_agumented.concatenate(train_dataset.map(process_translate))
133 data_agumented = data_agumented.concatenate(train_dataset.map(process_crop))
134
135 data_augmented = data_agumented.cache()
136 data_agumented = data_agumented.shuffle(buffer_size = 81600)
137 data_agumented = data_agumented.prefetch(buffer_size = AUTOTUNE)
```

Listing 3.4: Massive Data Augmentation

Após a aplicação destas funções, foi obtida a seguinte amostra de imagens:



Figura 3.2: Imagens de treino após *massive data augmentation*

Esta abordagem aumenta consideravelmente o tamanho do conjunto de treino e fornece ao modelo um conjunto rico de exemplos diversos para aprender.

### 3.4 GTSRB\_V4: Stacked Ensemble

De forma a experimentar a eficácia de um *ensemble*, foram unidos os modelos treinados no 3 *notebooks* anteriores num único *ensemble*. Para este efeito, foi criado um modelo que recebe como *input* as previsões (em concreto, os *logits*) dos modelos anteriores e aprende a classificar os sinais de trânsito através dessas previsões em vez de diretamente a partir das imagens.

A estrutura da rede neuronal treinada com estas previsões é a seguinte:

```

138 stack_model = tf.keras.models.Sequential([
139     tf.keras.layers.Flatten(input_shape=(len(train_logits_preds[0]),)),
140
141     tf.keras.layers.Dense(256),
142     BatchNormalization(), LeakyReLU(alpha=0.01),
143     tf.keras.layers.Dropout(0.4),
144     tf.keras.layers.Dense(128),
145     BatchNormalization(), LeakyReLU(alpha=0.01),
146     tf.keras.layers.Dropout(0.4),
147     tf.keras.layers.Dense(64),
148     BatchNormalization(), LeakyReLU(alpha=0.01),
149     tf.keras.layers.Dropout(0.4),
150
151     tf.keras.layers.Dense(43, activation='softmax')
152 ])

```

Listing 3.5: Stacked Ensemble NN Layers

# Capítulo 4

## Resultados obtidos

### 4.1 Notebook 1

Nesta primeira versão do modelo, sem qualquer tipo de processamento de dados, foi obtida uma **accuracy** de **97.514%**. Para além disto, foram usadas as funções `plot_predictions` e `show_missclassified` para observar de melhor forma as previsões que o modelo efetuou.

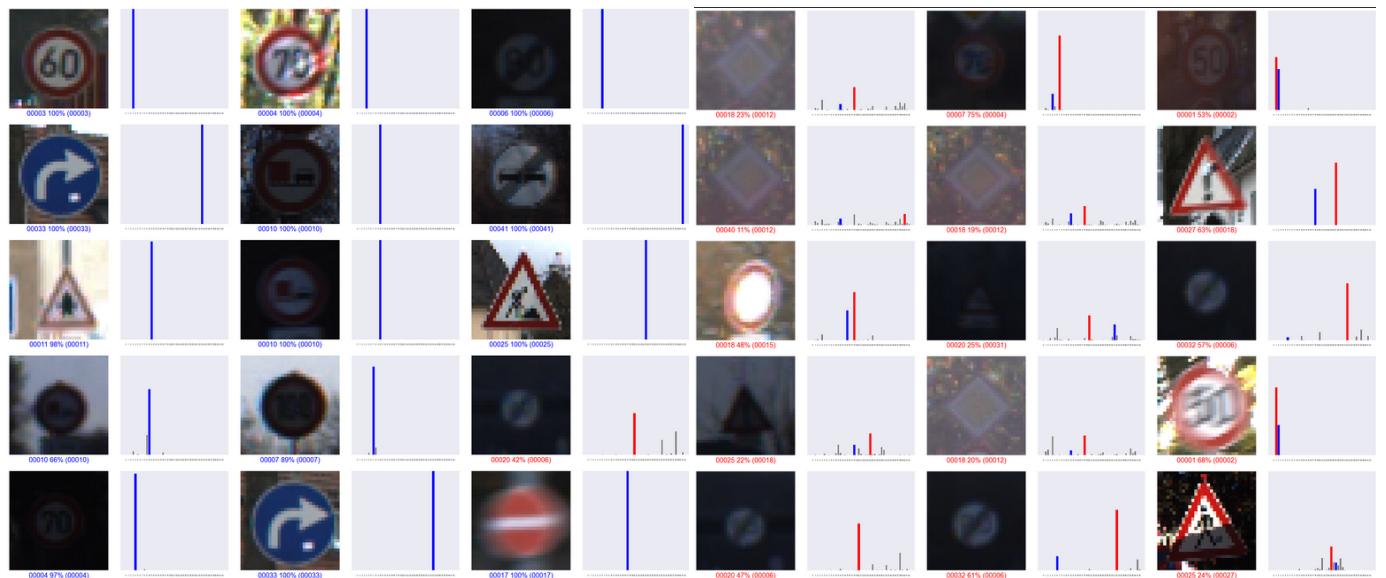


Figura 4.1: Plot\_Predictions

Figura 4.2: Show\_Missclassified

Obtendo-se também a seguinte matriz de confusão:

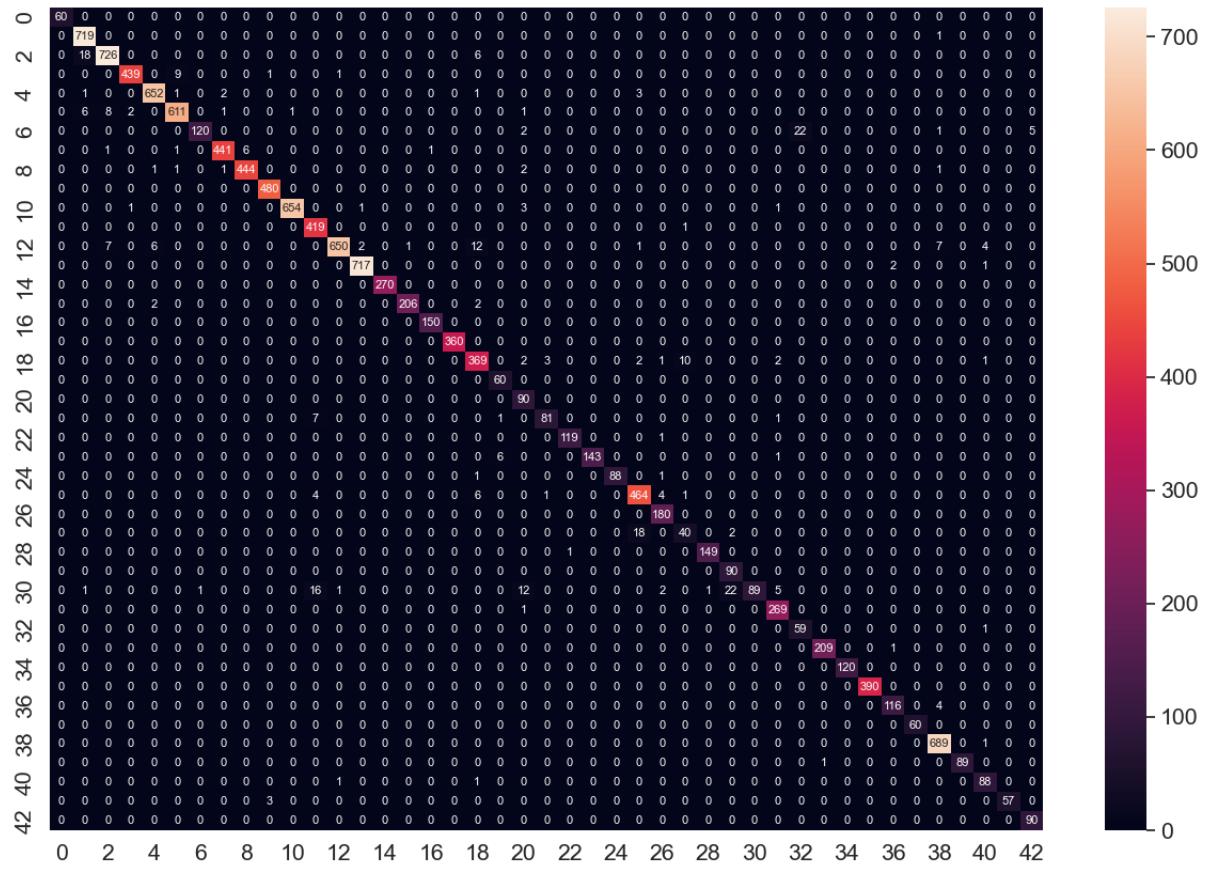


Figura 4.3: Matriz de Confusão

## 4.2 Notebook 2

Na segunda iteração do modelo, obeteve-se uma ***accuracy*** de **97.815%**. Embora marginal, foi observado um aumento ligeiro da *accuracy* do modelo em comparação à primeira iteração. Tal como anteriormente, foi ilustrado o desempenho do modelo com os seguintes gráficos:



Figura 4.4: Plot\_Predictions

Figura 4.5: Show\_Missclassified

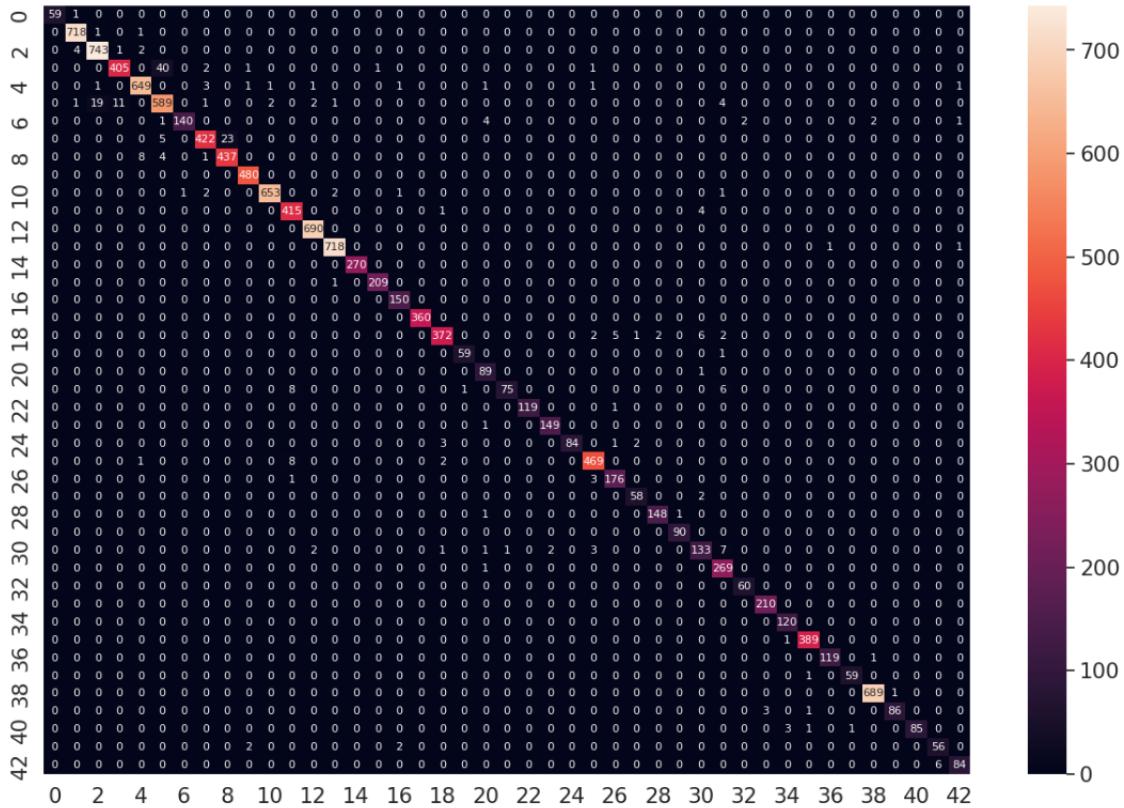


Figura 4.6: Matriz de Confusão

### 4.3 Notebook 3

Na terceira iteração do modelo, foi obtida uma **accuracy** de **99.224%**. Novamente, foi observado um maior aumento neste valor face às iterações anteriores. Os gráficos de desempenho obtidos são os seguintes:



Figura 4.7: Plot\_Predictions

Figura 4.8: Show\_Missclassified

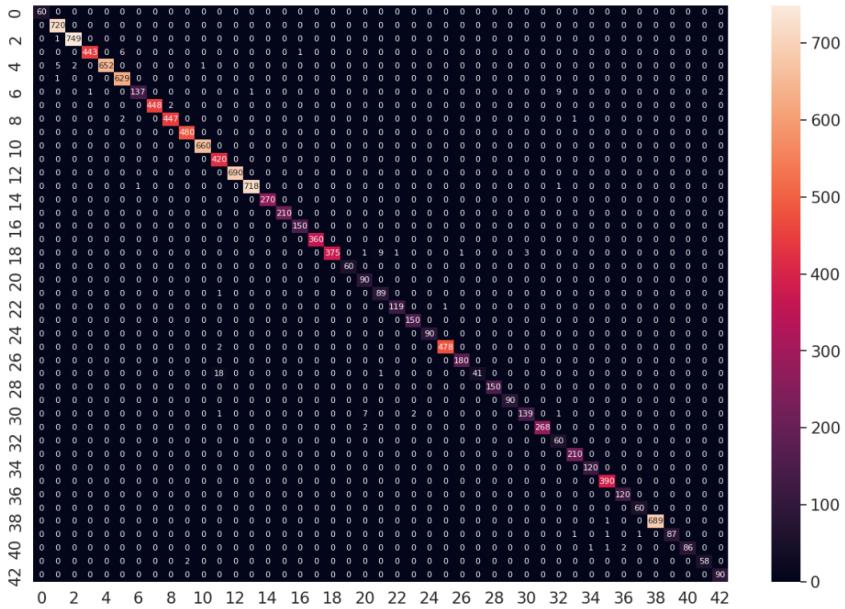


Figura 4.9: Matriz de Confusão

## 4.4 Notebook 4

Finalmente, para o modelo de *stacked ensemble*, foi obtida uma **accuracy de 98.583%**. Concluiu-se então que a forma como se constrói este modelo em *stacked ensemble* não resultou num modelo superior ao melhor modelo utilizado como modelo intermédio.

## 4.5 Conclusões e análise

Observou-se nas matrizes de confusão que uma parte considerável dos erros de previsão ocorrem em cenários específicos (prevê *A*, real *B*), em vez de estarem espalhados pela matriz inteira. Consultando as classes problemáticas, verificou-se que estes cenários muitas vezes representam sinais parecidos. Isto é, as duas classes representam sinais com símbolos semelhantes, cores semelhantes ou formas iguais.

Reparou-se também, nos restantes gráficos, que as previsões corretas são geralmente executadas com um enorme grau de confiança, enquanto que as previsões incorretas demonstram probabilidades menos confiantes e divididas entre várias classes.

Relativamente aos valores de *accuracy* obtidos, embora as mudanças nos valores sejam, à superfície, pequenas, neste patamar de *performance* este tipo de diferenças é significante. Conclui-se então que o método de *massive data augmentation* foi consideravelmente benéfico ao treino da rede neuronal utilizado, resultando num valor de *accuracy* bastante satisfatório.

Adicionalmente, embora o modelo de *stacked ensemble* tenha resultado num valor de *accuracy* algo menor à terceira iteração do projeto, é de notar que este utiliza uma rede neuronal consideravelmente mais simples que as utilizadas nas restantes iterações.

# Capítulo 5

## Conclusão Final

A equipa de desenvolvimento encontra-se satisfeita com o trabalho realizado e considera que se atingiu as metas de aprendizagem do mesmo. Foi aplicada a experiência adquirida ao longo do semestre para testar e comparar várias técnicas de treino de redes neurais, e estamos satisfeitos com os resultados finais.

Para trabalho futuro, foi considerado que há espaço no projeto para explorar e aprofundar a otimização da estrutura das redes neurais e dos hiperparâmetros das mesmas. Adicionalmente, a experimentação de mais métodos de desenvolvimento de modelos em *stacked ensemble* poderia ter fornecido resultados mais interessantes em contraste aos primeiros três modelos desenvolvidos, embora a terceira iteração do modelo já tenha uma *performance* difícil de melhorar consideravelmente. Por último, também existe a possibilidade de criar um *dataset* sintético de dados de validação, possivelmente isto ajudaria a ter dados de validação mais corretos e precisos que poderiam ajudar na melhoria da *performance* de todos os modelos.