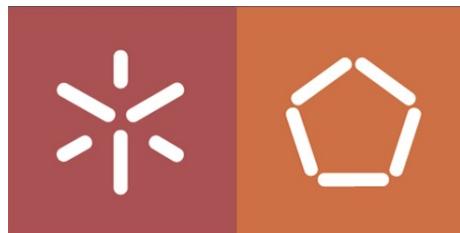


UNIVERSIDADE DO MINHO

DEPARTAMENTO DE INFORMÁTICA



MEI - MESTRADO EM ENGENHARIA INFORMÁTICA

COMPUTAÇÃO GRÁFICA

Visão por Computador e Processamento de Imagem

GRUPO 4

TRABALHO PRÁTICO 1



André Araújo PG47842



Diana Ferreira PG46529



Nuno Mata PG44420

29 de Maio
2021/2022

Conteúdo

1	Introdução	2
2	Contextualização	3
3	Implementação	4
3.1	Modelo	4
3.1.1	Metodologia Aplicada	4
3.1.2	Modelação e <i>Tuning</i>	4
3.2	Data Augmentation	5
3.2.1	Dynamic Data Augmentation	6
3.2.2	Massive Data Augmentation	8
3.3	Ensembles	9
4	Testes e Resultados Obtidos	10
4.1	Sem Data Augmentation	10
4.2	Dynamic Data Augmentation	11
4.2.1	Dynamic Data Augmentation versão 1	11
4.2.2	Dynamic Data Augmentation versão 2	12
4.2.3	Dynamic Data Augmentation versão 3	13
4.2.4	Comparação entre diferentes versões Dynamic Data Augmentation	13
4.3	Massive Data Augmentation	14
4.3.1	Massive Data Augmentation versão 1	14
4.3.2	Massive Data Augmentation versão 2	15
4.3.3	Comparação das duas versões de Massive Data Augmentation	15
4.4	Sem Data Augmentation <i>versus</i> Dynamic <i>versus</i> Massive Data Augmentation	16
4.5	Ensembles	16
5	Conclusão	17

1 Introdução

Este relatório surge no âmbito da Unidade Curricular **Visão por Computador e Processamento de Imagem** do mestrado de Engenharia Informática que está inserido no perfil **Computação Gráfica**.

O documento diz respeito ao primeiro trabalho prático da UC e pretende estudar a aplicação de modelos de *deep learning* aplicados ao *dataset* alemão de imagens de trânsito **GTSRB**, com o objetivo de implementar um modelo capaz de obter o melhor resultado possível nas suas previsões.

Para tal, inicialmente vão ser usadas técnicas de *data augmentation* e pré-processamento, que através da manipulação das imagens do *dataset* podem garantir melhorias mais tarde na performance do modelo. Por fim são ainda implementados métodos de *ensemble*, que usam vários algoritmos de aprendizagem para possibilitar a melhoria do modelo nas suas previsões.

Será também essencial, na parte final do documento, a apresentação de dados e a comparação dos resultados obtidos durante as várias fases de implementação do modelo para uma melhor compreensão do impacto das várias técnicas que foram sendo introduzidas ao longo do desenvolvimento do modelo final.

2 Contextualização

O modelo a ser desenvolvido para o trabalho prático 1 vai produzir previsões referentes ao *dataset* fornecido , o GTSRB(German Traffic Sign Recognition Benchmark). Este *dataset* vem dividido em dois ficheiros principais, um com o conjunto de imagens que vão ser usadas para treinar o modelo, e outro com o conjunto de imagens que vão ser utilizadas para testar/validar as previsões resultantes do modelo treinado.

Antes de iniciar a implementação do modelo existem parâmetros que podemos otimizar, à partida, que podem levar a uma melhoria posterior na sua performance, tais como o tratamento do *dataset*. Para este caso, como o *dataset* é apenas constituído por imagens e não existe praticamente mais informação fornecida sobre o conteúdo de cada imagem, as técnicas de tratamento de dados que podemos aplicar tornam-se mais limitadas (e.g. *Feature Engineering*: não faz sentido criar novas *features* a partir da relação de parâmetros/*features* existentes já que não conhecemos estes dados). No entanto foram aplicados métodos de *data augmentation*, que conseguem alterar "sinteticamente" o aspeto das imagens que vão ser usadas para treinar o modelo, de forma que ele se adapte a imagens mais adequadas àquelas que pode encontrar no mundo real. Com esta técnica as imagens sofrem alterações (ao nível de: orientação, localização, escala, cor, brilho, etc.) que garantem que o modelo vai ser treinado com um conjunto de imagens que não foram produzidas em ambientes semelhantes.

Relativamente aos métodos de aprendizagem desenvolvidos, foi desenvolvido um modelo baseado em redes neurais e também implementado um método de *Ensemble* que vai permitir a combinação de várias previsões de diferentes modelos na produção de resultados.

Por fim, é analisada a performance dos resultados obtidos para comparar quais técnicas de *Data Augmentation* e modelos de aprendizagem apresentaram melhores resultados.

3 Implementação

Neste capítulo é retratado todo o processo de implementação efetuado, nomeadamente, a descrição e elaboração do modelo baseado em *Convolutional Neural Network* (CNN), a apresentação das diversas aplicações de *Data Augmentation* utilizadas para treinar o modelo e, por último, a utilização de *ensembles* de redes, que faz uso das redes anteriormente treinadas.

3.1 Modelo

Este subcapítulo tem como foco principal descrever o modelo desenvolvido, referenciando as suas características, as várias ferramentas e bibliotecas aplicadas durante o seu desenvolvimento e, por último, especificando todos os hiperparâmetros utilizados para *tuning*.

3.1.1 Metodologia Aplicada

Em todo o processo foi utilizada a linguagem de *script python*, recorrendo-se, ainda, a ferramentas imprescindíveis para auxílio e facilitismo, como bibliotecas e documentação. Com o propósito de maximizar o processo de desenvolvimento do projeto, foram utilizadas bibliotecas direcionadas a *Machine Learning* (ML), que integram algoritmos e ferramentas cruciais para a implementação de redes neurais. Desta forma, foi utilizado o *tensorflow*, que permite ter um alto nível de abstração das ferramentas de ML por meio da *API (Application Programming Interface) Keras*.

3.1.2 Modelação e Tuning

Devido à complexidade do problema e à familiarização com o método escolhido, optamos desde início pela utilização de redes neurais. Dado que o problema é constituído por imagens, a nossa primeira abordagem consistiu na implementação de uma *Convolutional Neural Network*. Esta é definida como uma classe de *artificial neural network* (ANN) do tipo *feed-forward* e é aplicada com sucesso no processamento e análise de imagens digitais. Na concepção da CNN foram aplicadas três *Convolutional Layers*, seguidas de *Normalization Layers* e *MaxPooling Layers*. Posteriormente, aplicamos uma *Flattened Layer*, que tal como o nome indica "flattens" o input e uma camada densa. Por último, foi adicionada mais uma camada densa (camada de *output*), que contém o número de neurónios igual ao número de classes, 43, e uma função de ativação *softmax*.

Como já havia sido mencionado anteriormente, recorremos ao *Tensorflow* e à biblioteca *Keras* durante o desenvolvimento do modelo, sendo aplicados/implementados "elementos" particulares das redes neurais. Assim, este modelo contém vários hiperparâmetros que definem a estrutura ou topologia. Estes parâmetros incluem número de *hidden Layers*, número de neurónios por camada, função de ativação, número de *epochs*, função de *loss* e otimizador. É facilmente observável que o ajuste manual de todos estes parâmetros consome bastante tempo. É de notar também que os hiperparâmetros foram ajustados consoante a *validation-accuracy*, de modo a prevenir *overfitting*, isto é, o nosso modelo é generalizado e tem boa performance mesmo em dados desconhecidos.

O modelo final, como se pode observar na Tabela 1 e Figura 12, possui:

- Três *Convolutional Layers* 2D, com número de neurónios por camada que varia entre 64 a 256. A segunda e terceira *Convolutional Layer* são procedidas por uma função de ativação *LeakyReLU*, por uma *MaxPooling Layer*, uma *Normalization Layer* e por um *dropout* no valor de 0.5. A primeira *Convolutional Layer* é semelhante às posteriores, no entanto não é procedida por uma *MaxPooling Layer*.
- Uma *Flatten Layer*.
- Duas camadas densas. A primeira contém 256 neurónios e é seguida por uma função de ativação *LeakyReLU* e por um *dropout* no valor de 0.5. Já a segunda (camada de *output*) contém número de neurónios igual ao número de classes e uma função de ativação *softmax*.
- A função de *Loss* utilizada é a *categorical crossentropy*.
- O otimizador utilizado é *Adam* com *learning rate* de 0.0001.

Hiperparâmetro	Valor
Número de Convolutional Layers	3
Número de Normalization Layers	3
Número de MaxPooling Layers	2
Número de Flatten Layers	1
Número de Camadas Densas	2
Número de Neurônios	64:256 (por camada)
Função de Ativação	LeakyReLU, Softmax
Função de Loss	categorical crossentropy
Número de Epochs	10:30
Optimizer	Adam
Learning Rate	0.0001

Tabela 1: Hiperparâmetros do Modelo CNN.

```
def model_VI(classCount, imgSize, channels):
    model = Sequential()

    model.add(Conv2D(64, (5, 5), input_shape=(imgSize, imgSize, channels)))
    model.add(LeakyReLU(alpha=0.01))
    model.add(BatchNormalization())
    model.add(Dropout(0.5))

    model.add(Conv2D(128, (5, 5)))
    model.add(LeakyReLU(alpha=0.01))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(BatchNormalization())
    model.add(Dropout(0.5))

    model.add(Conv2D(256, (5, 5) ))
    model.add(LeakyReLU(alpha=0.01))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(BatchNormalization())
    model.add(Dropout(0.5))

    model.add(Flatten())
    model.add(LeakyReLU(alpha=0.0))
    model.add(Dense(256))
    model.add(LeakyReLU(alpha=0.0))
    model.add(Dropout(0.5))

    model.add(Dense(classCount, activation='softmax'))

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

Figura 2: Modelo CNN.

3.2 Data Augmentation

Após a conceção do modelo anteriormente mencionado, procedeu-se com o treino de vários modelos utilizando *Data Augmentation* diferentes, tanto em pré-processamento como em dinâmico. Desta forma, recorreu-se à utilização de vários métodos de processamento de imagens. Este subcapítulo tem como foco principal a apresentação dos várias tipos de *Data Augmentation* implementados e utilizados para treinar os modelos.

De forma a conseguir implementar os métodos de *Data Augmentation* foi necessário o uso das seguintes bibliotecas/módulos:

- **tf.image**
- **tfa.image**
- **tf.clip_by_value**

Estes módulos contêm várias funções de pré-processamento de imagem que vão ser usadas para realizar alterações ao nosso *dataset*. Por exemplo, através do uso do módulo *tf.image* foi possível manipular parâmetros das imagens como o brilho, com a função *tf.image.random_brightness*, o contraste, a tonalidade e a saturação (usando outras funções disponíveis através do módulo). Relativamente à segunda biblioteca, o *tfa.image*, esta deu acesso a funções que tornam possível a realização de rotações nas imagens, com a função *tfa.image.rotate*, a transformação das imagens, que podia ser expresso com resultados como imagens esticadas ou "off-centered" comparadas com a original

(dependendo dos parâmetros indicados na função) e por fim a alteração da escala de cores das imagens, que pode produzir imagens com cores muito diferentes das originais, podendo ser benéfico para o modelo.

3.2.1 Dynamic Data Augmentation

No que diz respeito ao *Dynamic Data Augmentation*, foram implementadas três versões diferentes. Na primeira versão efetuada, Versão 1, decidimos usufruir do material disponibilizado pelo docente da unidade curricular, ou seja, do código da matéria em questão. Desta forma, através das bibliotecas e módulos mencionados anteriormente, utilizamos para o processamento das imagens a rotação, translação, saturação e tonalidade. Como é possível observar na Figura 3, fizemos uso na translação e rotação do *tf.random.uniform*, que gera valores aleatórios de uma distribuição uniforme dentro do intervalo de -0,75 a 0,75 para a rotação e -3 a 3 para a translação. Para além disso, no caso da saturação e tonalidade, utilizamos o *tfa.image.random_hsv_in_yiq* de modo a ajustar a tonalidade, saturação e o valor de uma imagem RGB aleatoriamente no espaço de cores YIQ. Na Figura 4 é possível observar um conjunto de imagens processadas com esta implementação.

```
def process_image(image, label):

    # random rotate 5 degrees
    r = tf.random.uniform(shape=(), minval=-0.175, maxval=0.175, dtype=tf.dtypes.float32)
    image = tfa.image.rotate(image, r)

    # translate image up to 10%
    rx = tf.random.uniform(shape=(), minval=-3, maxval=3, dtype=tf.dtypes.float32)
    ry = tf.random.uniform(shape=(), minval=-3, maxval=3, dtype=tf.dtypes.float32)
    image = tfa.image.translate(image, [rx, ry])

    # change hue, saturation and value
    image = tf.clip_by_value(tfa.image.random_hsv_in_yiq(image, 0.2, 0.4, 1.1, 0.4, 1.1), 0, 1)

    return image, label
```

Figura 3: Dynamic Data Augmentation: Implementação da Versão 1.



Figura 4: Dynamic Data Augmentation: Imagens obtidas na Versão 1.

Após esta implementação, treinamos o modelo com esta versão de Dynamic Data Augmentation (Versão 1). Os resultados obtidos (*accuracy* e *loss*) vão ser apresentados no próximo capítulo, nomeadamente Testes e Resultados Obtidos.

No que diz respeito à Versão 2 de Dynamic Data Augmentation, o processamento de imagem baseia-se essencialmente no *lighting*. Decidimos optar por esta abordagem na Versão 2, visto que ao observar as imagens originais, ou seja, sem Data Augmentation ou qualquer tipo de processamento, notamos a existência de algumas imagens muito escurecidas. Desta forma, utilizamos para processamento das imagens o brilho, contraste, tonalidade e saturação. Todos estes quatro "elementos" foram aplicados de forma *random*. A implementação da Versão 2 é possível ser observada na Figura 5. Na Figura 6 é apresentado um conjunto de imagens processadas com esta implementação.

```
def process_image_lighting(image,label): #Random augmentation related to lighting
    image = tf.image.random_brightness(image, 1,) #apply random brightness
    image = tf.image.random_contrast(image, 1, 2,) #apply random contrast
    image = tf.image.random_hue(image, 0) #apply random hue
    image = tf.image.random_saturation(image, 1, 2,) #apply random saturation
    return image,label
```

Figura 5: Dynamic Data Augmentation: Implementação da Versão 2.

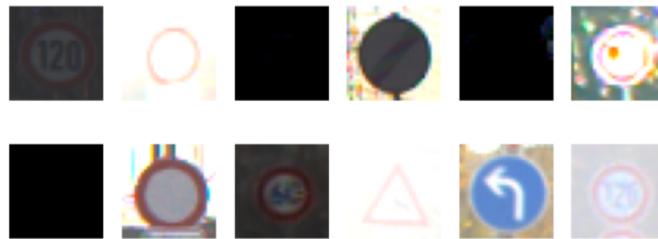


Figura 6: Dynamic Data Augmentation: Imagens obtidas na Versão 2.

Após esta implementação, treinamos o modelo aplicando esta versão de Dynamic Data Augmentation (Versão 2). Os resultados obtidos (*accuracy* e *loss*) vão ser apresentados no próximo capítulo, nomeadamente Testes e Resultados Obtidos.

Por último, efetuamos uma Versão 3 que foi elaborada com o intuito de obtermos uma ideia de como as cores RGB influenciam positivamente os resultados do treino do modelo. Desta forma, utilizamos o módulo *tf.image.rgb_to_grayscale* para efetuar a conversão de uma ou mais imagens de RGB para *Grayscale*. A implementação da Versão 3 é possível ser observada na Figura 7. Na Figura 8 é apresentado um conjunto de imagens processadas com esta implementação.

```
def process_image_gray(image,label,keep_channels=True):
    image = tf.image.rgb_to_grayscale(image)
    if keep_channels:
        image = tf.tile(image, [1, 1, 3])
    return image,label
```

Figura 7: Dynamic Data Augmentation: Implementação da Versão 3.



Figura 8: Dynamic Data Augmentation: Imagens obtidas na Versão 3.

Após a implementação da Versão 3, procedemos ao treinamento do modelo. Tal como já havia sido mencionado, os resultados obtidos (*accuracy* e *loss*) vão ser apresentados no próximo capítulo, nomeadamente Testes e Resultados Obtidos.

3.2.2 Massive Data Augmentation

No que diz respeito ao *Massive Data Augmentation*, foram implementadas duas versões diferentes (Versão 1 e Versão 2). Inicialmente, procedeu-se com a implementação dos vários métodos de processamento de imagem utilizados em ambas as versões, visto que vários são comuns. Tal como no caso do *Dynamic Data Augmentation*, decidimos usufruir do material disponibilizado pelos docentes da unidade curricular, mais especificamente o código referente a *Massive Data Augmentation*. Desta forma, através das bibliotecas e módulos previamente mencionados, implementamos os métodos apresentados na Figura 9. Ainda sobre a Figura 9, é possível observar a existência de métodos de processamento de imagem referentes à Cor, nomeadamente *Gray Scale*, saturação, brilho, contraste e tonalidade, que também foram utilizados no *Dynamic Data Augmentation*. Para além disso, é possível visualizar também métodos de processamento de imagem referentes à rotação, translação, *shear* e *crop*. Destes últimos, apenas o *shear* e *crop* não foram implementados também no *Dynamic Data Augmentation*. Em relação ao método de processamento de imagem *crop*, este extrai *crops* do *input image tensor* e redimensiona-os. No que diz respeito ao método de processamento de imagem *shear*, este aplica as transformações definidas a uma ou mais imagens.

```

def process_image_gray(image,label,keep_channels=True):
    image = tf.image.rgb_to_grayscale(image)
    if keep_channels:
        image = tf.tile(image, [1, 1, 3])
    return image,label

def process_brightness(image, label):
    img = tf.clip_by_value(tfa.image.random_hsv_in_yiq(image, 0.0, 1.0, 1.0, 0.1, 3.0),0,1)
    return img, label

def process_saturation(image, label):
    img = tf.clip_by_value(tfa.image.random_hsv_in_yiq(image, 0.0, 1.0, 3.0, 1.0, 1.0),0,1)
    return img, label

def process_contrast(image, label):
    img = tf.clip_by_value(tf.image.random_contrast(image, lower=0.1, upper=3.0, seed=None), 0, 1)
    return img, label

def process_hue(image, label):
    img = tf.image.random_hue(image, max_delta=0.2, seed=None)
    return img, label

def process_rotate(image, label):
    img = tfa.image.rotate(image, tf.random.uniform(shape=(), minval=-0.175, maxval=0.175))
    return img, label

def process_shear(image, label):
    img = tfa.image.rotate(image, tf.random.uniform(shape=(), minval=-0.175, maxval=0.175))
    sx = tf.random.uniform(shape=(), minval=-0.1, maxval=0.1, dtype=tf.dtypes.float32)
    img = tfa.image.transform(img, [1, sx, -sx*32, 0,1,0, 0,0])
    return img, label

def process_translate(image, label):
    img = tfa.image.rotate(image, tf.random.uniform(shape=(), minval=-0.175, maxval=0.175))
    tx = tf.random.uniform(shape=(), minval=-3, maxval=3, dtype=tf.dtypes.float32)
    ty = tf.random.uniform(shape=(), minval=-3, maxval=3, dtype=tf.dtypes.float32)
    img = tfa.image.translate(img, [tx,ty])
    return img, label

def process_crop(image, label):
    c = tf.random.uniform(shape=(), minval=24, maxval=30, dtype=tf.dtypes.float32)
    img = tf.image.random_crop(image, size=[c,c,3])
    img = tf.image.resize(img ,size= [30,30])
    return img, label

```

Figura 9: Massive Data Augmentation: Métodos de Processamento de Imagem.

Em relação à Versão 1, foram utilizados todos os métodos de processamento de imagem mencionados e apresentados na Figura 9, à exceção do *Gray Scale*. Relativamente à Versão 2, foram utilizados todos os métodos. Na Figura 10 e Figura 11 são expostas as imagens obtidas na Versão 1 e na Versão 2, respetivamente. É de notar a presença de imagens processadas com *Gray Scale* apenas na Versão 2, visto que só esta tem o respetivo método implementado.



Figura 10: Massive Data Augmentation: Imagens obtidas na Versão 1.



Figura 11: Massive Data Augmentation: Imagens obtidas na Versão 2.

3.3 Ensembles

Os métodos de *ensemble* são técnicas que criam vários modelos de aprendizagem e depois os combinam para produzir melhores resultados, geralmente produzem resultados mais precisos do que um único modelo, sendo assim, vai ser feita a sua implementação com o objetivo de testar se é possível ter melhorias na performance das previsões comparativamente com outros modelos que não usam este método. A implementação do modelo usado no *Ensemble* pode ser observada na Figura 12.

```
def create_model(classCount, imgSize, channels):
    modelLogits = Sequential()

    modelLogits.add(Conv2D(64, (5, 5), input_shape=(imgSize, imgSize, channels)))
    modelLogits.add(LeakyReLU(alpha=0.01))
    modelLogits.add(BatchNormalization())
    modelLogits.add(Dropout(0.5))

    modelLogits.add(Conv2D(128, (5, 5)))
    modelLogits.add(LeakyReLU(alpha=0.01))
    modelLogits.add(MaxPooling2D(pool_size=(2, 2)))
    modelLogits.add(BatchNormalization())
    modelLogits.add(Dropout(0.5))

    modelLogits.add(Conv2D(256, (5, 5)))
    modelLogits.add(LeakyReLU(alpha=0.01))
    modelLogits.add(MaxPooling2D(pool_size=(2, 2)))
    modelLogits.add(BatchNormalization())
    modelLogits.add(Dropout(0.5))

    modelLogits.add(Flatten())
    modelLogits.add(LeakyReLU(alpha=0.0))
    modelLogits.add(Dense(256))
    modelLogits.add(LeakyReLU(alpha=0.0))
    modelLogits.add(Dropout(0.5))

    modelLogits.add(Dense(classCount))

    output = Activation('softmax')(modelLogits.output)

    model = tf.keras.Model(modelLogits.inputs, output)

    opt = Adam(lr=0.0001)
    model.compile(optimizer = opt, loss='categorical_crossentropy', metrics=[ 'accuracy'])
    return model, modelLogits
```

Figura 12: Modelo CNN utilizado no Ensemble.

4 Testes e Resultados Obtidos

Neste capítulo são apresentados todos os testes efetuados e os respectivos resultados obtidos. Para além disso, são descritas análises sobre esses resultados obtidos e efetuadas comparações. Inicialmente, treinamos o modelo sem a aplicação de *Data Augmentation*, sendo que o resultado obtido e a avaliação do modelo é exposto no subcapítulo Sem Data Augmentation. Posteriormente, procedeu-se com o treino do modelo aplicando os diversos "tipos" (Versão 1, 2 e 3) de *Dynamic Data Augmentation* implementados. Por último, treinamos novamente modelos, mas agora aplicando dois tipos (Versão 1 e 2) de *Massive Data Augmentation*. Os resultados obtidos com a aplicação de *Data Augmentation* são apresentados no subcapítulo Dynamic Data Augmentation e Massive Data Augmentation, respectivamente.

4.1 Sem Data Augmentation

Como já havia sido mencionado, treinamos modelos sem *Data Augmentation* de modo a comparar os respetivos resultados obtidos com os resultados do treino de modelos aplicando vários tipos de *Data Augmentation*. O treino dos modelos foi efetuado com um total de 20 epochs. Após o treino, foi gerado o gráfico da Figura 13 e obtidos os resultados da Figura 14. Estes resultados são da avaliação (*accuracy* e *loss*) do modelo com base no dataset de teste e dataset de validação, respetivamente. No final deste subcapítulo, apresentamos um gráfico que compara os resultados obtidos a partir das diferentes versões.

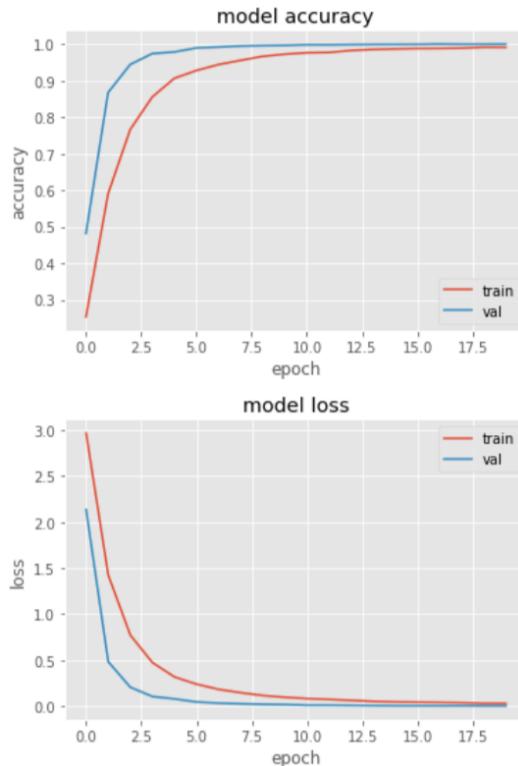


Figura 13: Gráfico de desempenho da accuracy e loss ao longo das epochs.

198/198 - 1s - loss: 0.0696 - accuracy: 0.9801 - 1s/epoch - 7ms/step
4/4 - 1s - loss: 0.0026 - accuracy: 1.0000 - 930ms/epoch - 233ms/step

Figura 14: Resultado final da accuracy e loss.

4.2 Dynamic Data Augmentation

Neste subcapítulo são apresentados os resultados obtidos relativamente ao treino de modelos aplicando as 3 versões de *Dynamic Data Augmentation*. O treino dos modelos foi efetuado com um total de 30 epochs. Nos próximos subsubcapítulos serão apresentados os gráficos gerados e os resultados obtidos após o treino aplicando as diferentes versões.

4.2.1 Dynamic Data Augmentation versão 1

Após o treino dos modelos aplicando a Versão 1 de *Dynamic Data Augmentation*, foi gerado o gráfico da Figura 15 e obtidos os resultados da Figura 16. Estes resultados são da avaliação (*accuracy* e *loss*) do modelo com base no dataset de teste e dataset de validação, respetivamente.

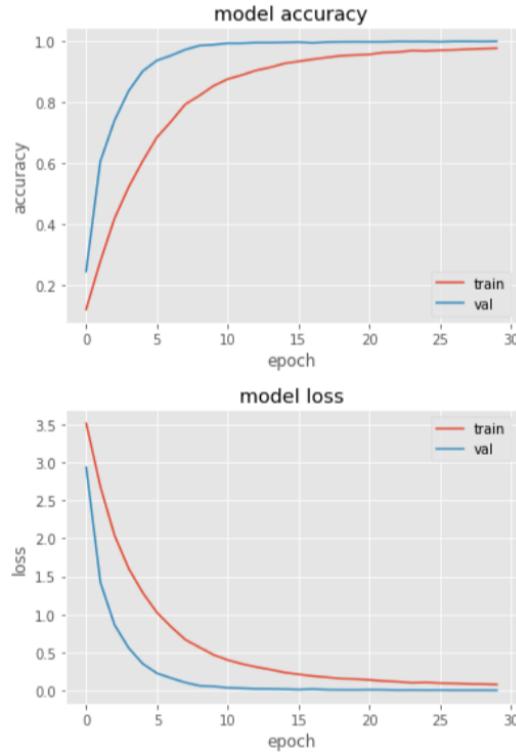


Figura 15: Gráfico de desempenho da accuracy e loss ao longo das epochs.

```
198/198 - 1s - loss: 0.0471 - accuracy: 0.9871 - 1s/epoch - 7ms/step
4/4 - 1s - loss: 0.0033 - accuracy: 0.9992 - 940ms/epoch - 235ms/step
```

Figura 16: Resultado final da accuracy e loss.

4.2.2 Dynamic Data Augmentation versão 2

Após o treino dos modelos aplicando a Versão 2 de *Dynamic Data Augmentation*, foi gerado o gráfico da Figura 17 e obtidos os resultados da Figura 18. Estes resultados são da avaliação (*accuracy* e *loss*) do modelo com base no dataset de teste e dataset de validação, respectivamente.

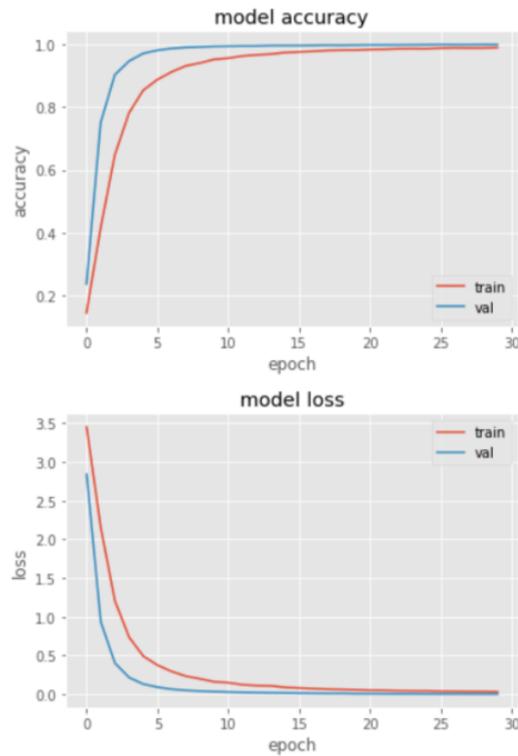


Figura 17: Gráfico de desempenho da accuracy e loss ao longo das epochs.

```
198/198 - 1s - loss: 0.0635 - accuracy: 0.9813 - 1s/epoch - 6ms/step
4/4 - 1s - loss: 0.0047 - accuracy: 0.9994 - 927ms/epoch - 232ms/step
```

Figura 18: Resultado final da accuracy e loss.

4.2.3 Dynamic Data Augmentation versão 3

Após o treino dos modelos aplicando a Versão 3 de *Dynamic Data Augmentation*, foi gerado o gráfico da Figura 19 e obtidos os resultados da Figura 20. Estes resultados são da avaliação (*accuracy* e *loss*) do modelo com base no dataset de teste e dataset de validação, respectivamente.

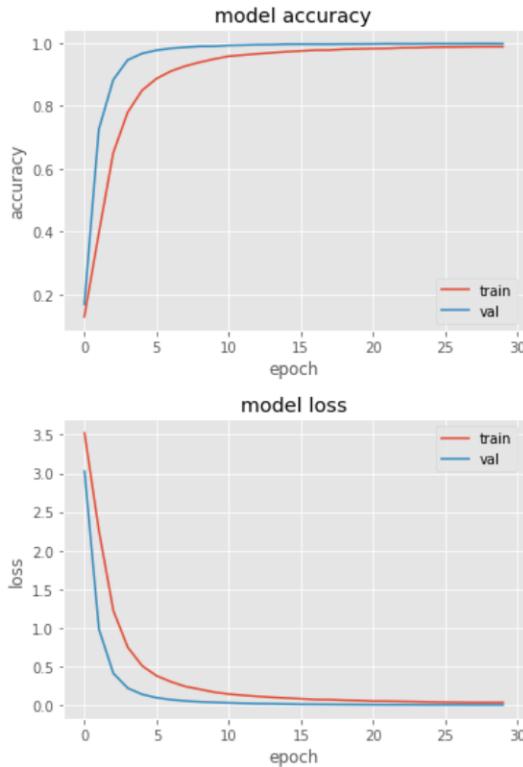


Figura 19: Gráfico de desempenho da accuracy e loss ao longo das epochs.

```
198/198 - 2s - loss: 0.0709 - accuracy: 0.9799 - 2s/epoch - 8ms/step
4/4 - 1s - loss: 0.0047 - accuracy: 0.9991 - 937ms/epoch - 234ms/step
```

Figura 20: Resultado final da accuracy e loss.

4.2.4 Comparação entre diferentes versões Dynamic Data Augmentation

Como é possível observar na Figura 21, a versão que originou melhores resultados, ou seja, teve mais sucesso quando aplicada no treino de modelos, foi a Versão 1, que na figura representa (V2). Na figura a Versão 2 está representada como (V2.1) e a Versão 3 como (V2.2). Quem obteve piores resultados entre as 3 versões foi a Versão 3, respetiva ao método de processamento de imagem *Gray Scale*, que converte uma ou mais imagens RGB em *Gray Scale*.

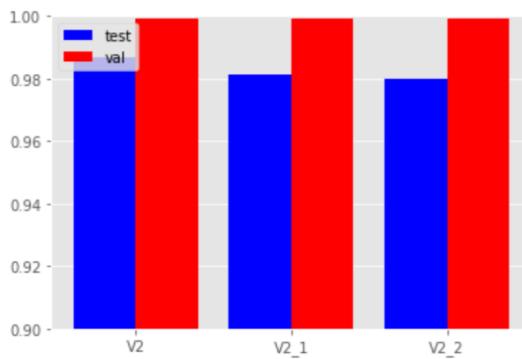


Figura 21: Comparação entre as redes treinadas com as diferentes versões de Dynamic Data Augmentation.

4.3 Massive Data Augmentation

Neste subcapítulo são apresentados os resultados obtidos relativamente ao treino de modelos aplicando as 2 versões de *Massive Data Augmentation*. O treino dos modelos foi efetuado com um total de 10 epochs. Nos próximos subsubcapítulos serão apresentados os gráficos gerados e os resultados obtidos após o treino aplicando as diferentes versões.

4.3.1 Massive Data Augmentation versão 1

Após o treino dos modelos aplicando a Versão 1 de *Massive Data Augmentation*, foi gerado o gráfico da Figura 22 e obtidos os resultados da Figura 23. Estes resultados são da avaliação (*accuracy* e *loss*) do modelo com base no dataset de teste e dataset de validação, respetivamente.

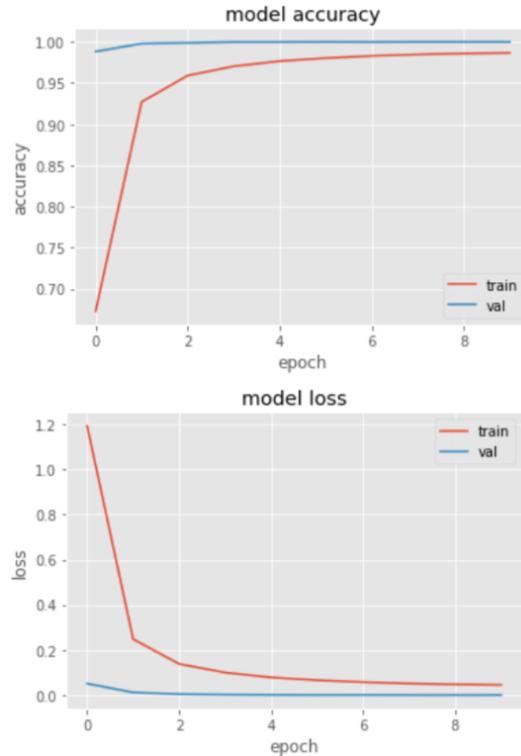


Figura 22: Gráfico de desempenho da accuracy e loss ao longo das epochs.

```
198/198 - 2s - loss: 0.0334 - accuracy: 0.9888 - 2s/epoch - 10ms/step
4/4 - 1s - loss: 7.1010e-04 - accuracy: 1.0000 - 1s/epoch - 316ms/step
```

Figura 23: Resultado final da accuracy e loss.

4.3.2 Massive Data Augmentation versão 2

Após o treino dos modelos aplicando a Versão 2 de *Massive Data Augmentation*, foi gerado o gráfico da Figura 24 e obtidos os resultados da Figura 25. Estes resultados são da avaliação (*accuracy* e *loss*) do modelo com base no dataset de teste e dataset de validação, respectivamente.

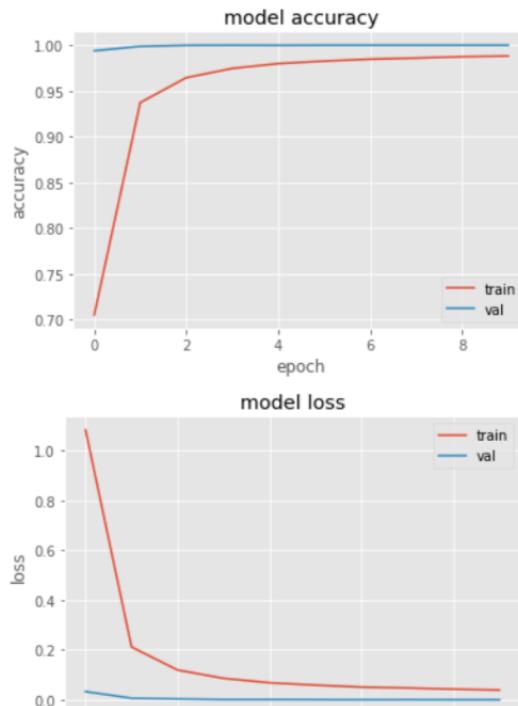


Figura 24: Gráfico de desempenho da accuracy e loss ao longo das epochs.

```
198/198 - 1s - loss: 0.0315 - accuracy: 0.9916 - 1s/epoch - 7ms/step
4/4 - 1s - loss: 0.0010 - accuracy: 1.0000 - 980ms/epoch - 245ms/step
```

Figura 25: Resultado final da accuracy e loss.

4.3.3 Comparaçāo das duas versões de Massive Data Augmentation

Como é possível observar nos dois sub-subcapítulos anteriores relativos às versões de *Massive Data Augmentation*, a versão que originou melhores resultados, ou seja, teve mais sucesso quando aplicada no treino de modelos, foi a Versão 2, onde foram aplicados todos os métodos de processamento de imagem, incluindo o de *Gray Scale*.

4.4 Sem Data Augmentation *versus* Dynamic *versus* Massive Data Augmentation

Este subcapítulo tem como foco principal analisar e comparar os resultados dos vários casos testados, ou seja, resultados do treino de modelos sem *Data Augmentation*, com *Dynamic Data Augmentation* e com *Massive Data Augmentation*. Como é possível observar na Figura 26, o caso que originou melhores resultados, ou seja, com uma melhor accuracy tendo como base o dataset de teste, foram os modelos treinados com a aplicação de *Massive Data Augmentation* (V3). Desta forma, podemos concluir que os métodos de processamento aplicados no *Massive Data Augmentation* originaram resultados bastante positivos, melhorando positivamente a previsão. Em contraste, o caso onde obtivemos piores resultados foi o do treino de modelos sem *Data Augmentation* (V1), onde o modelo é treinado com as imagens originais, que foram tiradas em circunstâncias diferentes.

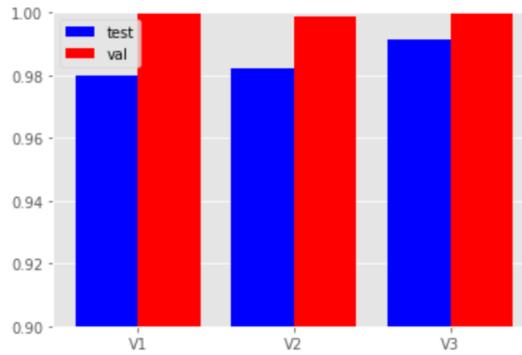


Figura 26: Comparação entre as redes treinadas com Dynamic Data Augmentation, Massive Data Augmentation e sem Data Augmentation.

4.5 Ensembles

Como já havia sido mencionado, *Ensembles* de rede são técnicas que criam vários modelos de aprendizagem e depois combinam-nos de modo a obter melhores resultados. Desta forma, efetuamos vários testes com cada caso (Sem *Data Augmentation*, *Dynamic Data Augmentation* e *Massive Data Augmentation* utilizando *Ensembles*, sendo um deles apresentado nas Figura 27 e Figura 28. O caso dessas figuras é referente ao de *Massive Data Augmentation*, sendo possível observar que os valores obtidos são o expectável, ou seja, dentro dos 99% de *accuracy*.

```
198/198 - 1s - loss: 0.0232 - accuracy: 0.9929 - 1s/epoch - 6ms/step
198/198 - 1s - loss: 0.0365 - accuracy: 0.9876 - 1s/epoch - 6ms/step
198/198 - 1s - loss: 0.0280 - accuracy: 0.9923 - 1s/epoch - 6ms/step
198/198 - 1s - loss: 0.0261 - accuracy: 0.9933 - 1s/epoch - 6ms/step
198/198 - 1s - loss: 0.0205 - accuracy: 0.9946 - 1s/epoch - 6ms/step
average accuracy: 99.213
```

Figura 27: Resultados da aplicação de Ensembles de redes com as redes de *Massive Data Augmentation*.

```
198/198 - 1s - loss: 0.0302 - accuracy: 0.9913 - 1s/epoch - 6ms/step
4/4 - 1s - loss: 3.9995e-04 - accuracy: 1.0000 - 980ms/epoch - 245ms/step
```

Figura 28: Resultados da aplicação de Ensembles de redes com as redes de *Massive Data Augmentation*.

5 Conclusão

Ao longo da realização do primeiro trabalho prático da UC conseguimos estudar a aplicação de modelos de *deep learning* aplicados ao *dataset* alemão de imagens de trânsito **GTSRB** e conseguimos implementar um modelo capaz de obter um resultado bastante satisfatório nas suas previsões.

Para a obtenção destes resultados, testamos a utilização de técnicas de *data augmentation* e pré-processamento, que através da manipulação das imagens do *dataset* para melhorar a performance do modelo, esta melhoria foi mais visível com a utilização do *Massive Data Augmentation*. Por fim com a implementação dos métodos de *ensemble* permitiu uma ligeira melhoria em relação às previsões anteriores.

Concluindo achamos que conseguimos testar a diversas técnicas de *data augmentation* e o uso de métodos de *ensemble* com sucesso, obtendo bons resultados, contudo não conseguimos ultrapassar o melhor resultado publicado de 99.81%, para tal poderíamos ter feito mais uso dos filtros e das convulsões aprendidas na secção de aquisição e processamento de imagem desta UC, bem como ter testado mais arquiteturas de *deep learning* nos nossos testes.