Universidade do Minho

Mestrado Integrado em Engenharia Informática

Computação Natural

2º semestre/4º ano

Classificação de Imagens com Redes Neuronais Convolucionais

a86268 Maria Pires

3 de maio de 2021

Conteúdo

Lista de Figuras

1 | Introdução

O presente relatório foi elaborado no âmbito da unidade curricular *Computação Natural*, inseridano perfil *Sistemas Inteligentes* e detalha o processo do trabalho prático individual.

O modelo `Convolutional Neural Networks (CNN)` é um algoritmo de *Deep Learning* que através de uma imagem define pesos para multiplos aspetos ou objetos desta.

A arquitetura de uma rede `CNN` tem dois principais objetivos: a extração de *features* e a agregação destas para efetuar uma previsão. Cada imagem recebida como *input* passa, por várias camadas de convolução como filtros, *pooling* e/ou redes com camadas completamente conectadas. A primeira camada executa a extração de *features*, isto é, aprende os principais atributos da imagem. Esta passo é fundamental pois a redução dimensional das imagens facilita o seu processamento. Contudo, é necessário que nesta redução não ocorra a perda de informação relevante. Assim, esta usa um filtro, executa a redução dimensional que torna possível a obtenção de cantos, manchas e propriedades de objetos após a aplicação de vários filtros. Na aplicação destes filtros, pode ser necessário a definição de parâmetros como *padding*. De seguida, pode existir uma camada de *Pooling* que tem o mesmo intuito das *convolutional layers*, ou seja, reduzir o número de parâmetros quando as imagens possuem uma grande dimensão.

Por último, é aplicada uma camada de *Flattening* e, de seguida, uma *Fully Connected Layer*. A primeira organiza os resultados dos *feature maps* obtendo um vetor que irá servir como *input* à *fully connected layer*. Nesta procedemos à classificação das imagens, pelo que se recorre a *backpropagation* e redistribuição dos pesos para a aprendizagem através da técnica de `softmax`, por exemplo.

Posto isto, pressupõem-se a criação de uma rede convolucional neuronal para classificar 250 espécies de pássaros e de um algoritmo genético com o objetivo de otimizar os vários parâmetros dessa rede. Neste relatório será exposto todo o processo de investigação, implementação e resultados obtidos.

2 | Análise e tratamento de dados

O conjunto de dados fornecido é composto por 250 espécies distintas de aves. Cada imagem presente no *dataset* tem dimensão 224x224 pixeis e três canais de cor.

O conjunto apresenta a seguinte distribuição:

- Imagens para treino: 35215
- Imagens para teste: 1250
- Imagens para validação: 1250

A distribuição das imagens de treino por classe encontra-se entre as 90 e 300 imagens. Quanto às imagens de teste e validação existem 5 fotografias para cada uma das 250 classes.

O único tratamento relevante nesta fase inicial foi a normalização das imagens. Será exposto ao longo do relatório a necessidade que surgiu e em que momentos de reduzir a dimensão das imagens e a aplicação de *data augmentation*. Esta última com o intuito de criar maior diversidade e tornar a rede mais robusta.

Figura 2.1: Exemplos de imagens presentes no conjunto de validação

3 | Redes Neuronais Convolucionais

O primeiro passo na resolução do trabalho foi a implementação de uma rede neuronal convolucional simples, com os seguintes parâmetros escolhidos aleatoriamente, apenas com o objetivo de fazer um primeiro teste:

- 3 camadas intermédias;
- 128 filtros nas camadas de convolução;
- Tamanho do Kernel (3,3);
- Valor de *padding* aplicado foi `same`;
- Aplicação de *MaxPooling2D* com o tamanho (2,2) nas camadas intermédias;
- Aplicação de *BatchNormalization* e *Dropout* com um valor de 0,25;
- Função de ativação `relu` para todas as camadas exceto a última na qual se aplicou `softmax`;
- A função de *Loss* aplicada foi `categorical_crosstentropy`;
- A métrica medida no treino foi apenas a *accuracy*;
- Nesta CNN manteve-se o *input_ shape* original de (224,224,3).

Tendo o conhecimento prévio do peso computacional associado ao treino da rede a construção desta rede simples permitiu não só ter uma noção inicial dos valores de treino que poderia ser esperados em iterações futuras mas também definir e compreender quais as métricas necessárias para avaliar os modelos que surgiriam após a aplicação do algoritmo genético. A rede obtida nesta primeira iteração é constituída por 13,326,330 parâmetros e não foi aplicada *data augmentation*. Foi mantida a *shape* original das imagens utilizadas para o treino desta rede (224,224,3).

Após o *fit* do modelo com 100 *epochs* e *early stopping* obteram-se os resultados expostos na tabela 3.1.

	CNN
Train Accuracy	0.9973
Train Loss	0.0400
Validation Accuracy	0.6480
Validation Loss	1.4994
Test Accuracy	0.6567
Test Loss	1.4369

Tabela 3.1: Valores obtidos após o treino da rede

Aplicando a função *classification_ report* do `Keras` ao conjunto de teste após a avaliação do modelo é possível analisar métricas como:

- *Recall*, que indica quantos elementos da classe foram encontrados tendo em conta o número de elementos da classe;
- *Precision*, que calcula quantos elementos foram classificados corretamente dentro da classe;
- *F1-Score* que representa a média harmonica entre a *Precision* e o *Recall*;
- *Support*, que indica o número de ocorrencias de cada classe no conjunto dados. O conjunto tem 5 imagens em cada classe pelo que em termos de distribuição de classes está perfeitamente balanceado.

Em média, para esta rede foi obtida uma *precision* de 0.7 e um recall de *0.66*, como se pode observar pela figura seguinte.

	precision	recall	f1-score	support
0	0.50	0.60	0.55	5
1	0.80	0.80	0.80	5
2	0.33	0.20	0.25	5
3	0.67	0.40	0.50	5
4	0.50	1.00	0.67	5
5	0.67	0.80	0.73	5
6	0.62	1.00	0.77	5
		(...)		
245	0.71	1.00	0.83	5
246	0.67	0.40	0.50	5
247	0.80	0.80	0.80	5
248	1.00	1.00	1.00	5
249	0.60	0.60	0.60	5
accuracy			0.66	1250
macro avg	0.70	0.66	0.65	1250
weighted avg	0.70	0.66	0.65	1250

Figura 3.1: Excerto da matriz de classificação da rede

Em complemento ao *Classification Report*, foi também imprimido para cada imagem presente no conjunto de dados a classe a que pertencia e qual tinha sido prevista pelo modelo. Pode observar-se algumas das classificações na figura seguinte.

```
Predicted a 4. Real value is 2.
Predicted a 82. Real value is 2.
Predicted a 2. Real value is 2.
Predicted a 155. Real value is 2.
Predicted a 4. Real value is 2.
Predicted a 87. Real value is 3.
Predicted a 196. Real value is 3.
Predicted a 3. Real value is 3.
Predicted a 198. Real value is 3.
Predicted a 3. Real value is 3.
Predicted a 4. Real value is 4.
Predicted a 4. Real value is 4.
Predicted a 4. Real value is 4.
Predicted a 4. Real value is 4.
Predicted a 4. Real value is 4.
```

Figura 3.2: Previsões do modelo para o conjunto de teste

Na figura 3.2 é possível verificar que as *features* da classe 4 foram aprendidas pela rede com maior eficácia que por exemplo as da classe 2. Na figura 3.1 podemos observar que o *recall* da classe 4 é 1.00, dentro da classe todas os elementos foram classificados corretamente, no entanto a sua precisão é apenas 0.50 pois há elementos que não pertencem à classe que foram classificados como tal.

3.1 Data Augmentation

Com o objeto de construir uma rede robusta o segundo passo foi aplicar *Data Augmentation* (DA) aquando do carregamento das imagens recorrendo à classe *ImageDataGenerator*. Esta técnica pode também ser aplicada através do modulo *tf.keras.layers.experimental.preprocessing*. Este modulo permite ter mais controlo sobre o processo pois é possível decidir em que camada especifica se vai aplicar *Data Augmentation*, contudo como o nome indica ainda é experimental pelo que foi utilizada a primeira opção.

Foram aplicadas as opções que se apresentam de seguida, a escolha destas foi aleatória. O algoritmo genético poderia também ser utilizado para descobrir quais os campos que otimizam o processo de aprendizagem da rede, contudo optei por utilizar estes parâmetros para todos os modelos testados ao longo do trabalho, com exceção das arquiteturas de *transfer learning*.

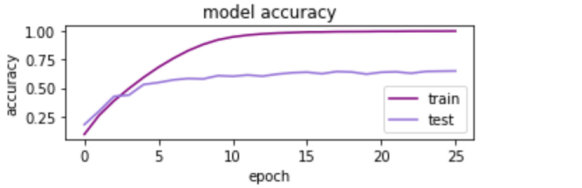
- rotation_range = 0.2
- shear_range = 0.5,
- horizontal_flip=True,
- vertical_flip=True

Utilizando o modelo apresentado anteriormente, mas alimentando o treino com imagens às quais foi aplicada *Data Augmentation* em vez do conjunto original verificou-se um incremento de cerca de 10% na *Accuracy* do conjunto de

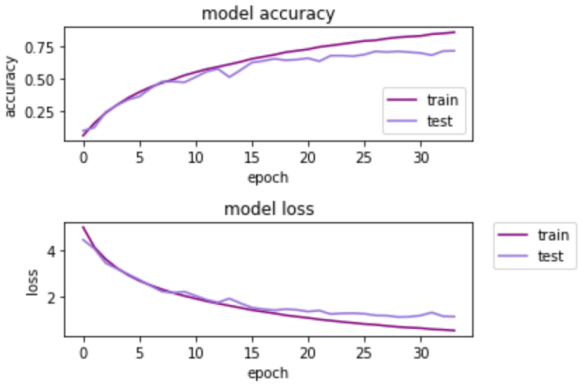
teste. Ao longo do treino verificou-se também uma redução do *overfitting* após a introdução de DA, tendo havido uma redução na *Accuracy* de treino de cerca de 15%.

	CNN	CNN + DA
Train Accuracy	0.9973	0.8583
Train Loss	0.0400	0.5240
Validation Accuracy	0.6480	0.7136
Validation Loss	1.4994	1.1335
Test Accuracy	0.6567	0.7544
Test Loss	1.4369	0.9925

Tabela 3.2: Resultados obtidos após o treino dos modelos



(a) CNN



(b) CNN + Data Augmentation

Figura 3.3: Variação da *Accuracy* e *Loss* ao longo do treino das redes

O código desenvolvido que suporta este capitulo pode ser consultado no ficheiro `CNN+DA224.ipynb`.

4 | Algoritmo Genético

4.1 Construção do Algoritmo Genético

Após a criação de uma rede neuronal convolucional simples, para automatizar a procura pelos pelos parâmetros que otimizam o desempenho da rede foi desenvolvido um algoritmo genético. Este algoritmo baseia-se no processo de seleção natural, no qual uma população é constituída por indivíduos e os mais aptos são selecionados com o objetivo de produzir a próxima geração. A nova geração herda as características com maior potencial de sobrevivência, o que neste caso se traduz nos hiper-parametros que melhoram a *performance* da rede. O processo de seleção é repetido ao longo das gerações até que se encontrem os melhores parâmetros. Posto isto, o algoritmo inicia-se pela construção de uma população, isto é, um conjunto de cromossomas constituído por genes. Cada cromossoma é uma solução para o problema.

4.1.1 Constituição dos cromossomas

O primeiro passo é escolha dos parâmetros da redes, ou seja, a definição dos genes. Apresentam-se de seguidas os parâmetros possíveis para cada gene:

- **Número de camadas intermédias:** entre 1 e 5;
- **Número de filtros:** 32, 64, 128, 256;
- **Função de Ativação:** `relu`, `selu`, `softmax`;
- **Função de Otimização:** `SGD`, `Adam`, `Adagrad`, `RMSprop`;
- **Função de Perda:** `categorical_crossentropy`, `categorical_hinge`;
- **Taxa de Aprendizagem:** 0.001, 0.01, 0.1;
- **Padding:** `same`, `valid`;
- **Dropout:** 0.15, 0.25, 0.3, 0.5.

Embora existam mais parâmetros que poderiam ser aplicados, como por exemplo o tamanho do kernel ser (5,5) ou (7,7) em vez de (3,3) definiram-se apenas os parâmetros apresentados anteriormente devido à elevada carga computacional inerente à utilização de um AG em conjunto com uma CNN.

Adicionalmente, poderiam também ser introduzidos no gene parâmetros como a adição de camadas com e sem *dropout* para otimizar a sua estrutura e não apenas os parâmetros apresentados. Contudo optei por manter uma estrutura fixa, isto é, embora o número de camadas intermédias seja dinâmico e

definido pelo algoritmo genético, por sua vez, parâmetros como *MaxPooling2D*, *BatchNormalization* e *Dropout* farão sempre parte de cada camada intermédia. Inicialmente, e analisando todos os componentes necessários para construir uma CNN a lista composta por 8 elementos apresentada anteriormente pareceu-me conter os constituintes mais básicos e mais importantes para o processo de otimização. Novamente, esta decisão foi motivada pela elevada carga computacional.

Na figura 4.1 pode observar-se a função genérica que cria uma rede com os genes definidos acima.

```
def createModel(cl, filters, aFunc, optFunc, lossFunc, lr, drop, pad):
    model = Sequential()
    # first convolution layer
    model.add(Conv2D(filters, kernel_size=(3,3), padding=pad, input_shape=shape, activation=aFunc))
    model.add(BatchNormalization())
    # middle convolutional layers
    for i in range(int(cl)):
        model.add(Conv2D(filters, kernel_size=(3,3), padding=pad, input_shape=shape, activation=aFunc))
        model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(BatchNormalization())
        model.add(Dropout(drop))
    # flattening
    model.add(Flatten())
    model.add(Dropout(drop))
    # full connection
    model.add(Dense(filters,activation='relu'))
    model.add(BatchNormalization())
    # dropout
    model.add(Dropout(drop))
     # output layer
    model.add(Dense(nSpecies,activation="softmax"))

    if(optFunc == 'SGD'): optFunc = SGD(learning_rate=lr)
    elif(optFunc == 'Adam'): optFunc = Adam(learning_rate=lr)
    elif(optFunc == 'Adagrad'): optFunc = Adagrad(learning_rate=lr)
    elif(optFunc == 'RMSprop'): optFunc = RMSprop(learning_rate=lr)

    #compile model
    model.compile(optFunc, loss=lossFunc, metrics=['accuracy'])

    return model
```

Figura 4.1: Função genérica de criação do modelo

4.1.2 Avalição dos cromossomas

Foi definida a seguinte função de *fitness* para definir a probabilidade de um cromossoma ser selecionado para a próxima geração:

$$score = 0.7 * accuracy + 0.15 * precision + 0.15 * recall$$

O objetivo do algoritmo é selecionar os pais que produzem uma melhor *accuracy* não descurando a importância de classificar corretamente a própria classe pelo que a *precision* e *recall* possuem um peso menor na decisão.

Idealmente, para obter os melhores resultados possíveis deveria ser produzida a maior diversidade possível pelo que a aplicação de *Cross Validation* teria sido uma mais valia neste processo de avaliação e validação dos cromossomas. Contudo isso traria uma carga adicional com a qual creio que não teria recursos para lidar.

4.1.3 Seleção

Nesta fase os cromossomas que apresentam o melhor *score* são selecionados para que os seus genes sejam passados à proxima geração.

4.1.4 Crossover

Na fase de *crossover* o filho é criado a partir da junção de metade dos cromossomas dos seus pais.

4.1.5 Mutação

Nesta fase, um dos genes do filho é aleatóriamente alterado com o objetivo de manter a diversidade e evitar a convergência prematura do modelo.

4.2 Processo de obtenção de resultados

Após a construção de um algoritmo genérico com as características mencionadas anteriomente foi necessário testar os seus resultados.

Numa primeira abordagem achei razoável utilizar a *shape* original das imagens, contudo após várias tentativas esta abordagem pareceu-me que não seria sustentável pois surgiam múltiplos problemas. Dado que as redes são construídas aleatoriamente e com uma quantidade de a parâmetros muito elevada estas esgotavam os recursos disponibilizados pelas plataformas utilizadas ou então eram demasiado demoradas nunca sendo possível completar mais do que uma ou duas gerações constituídas por uma população inicial de 5, 8 ou 10 indivíduos. Foram efetuados diversos testes com diferentes números de épocas, passos por época, tamanhos de população e número de gerações mas os problemas mantiveram-se ou sendo resolvidos não produziam resultados satisfatórios.

Posto isto, decidi reduzir a *shape* para (100,100,3) o que permitiu correr várias gerações e obter resultados mais sólidos. No entanto, com os recursos disponíveis continuava a não me ser possível correr várias gerações e os scores das combinações obtidas não eram muito satisfatórios pelo que defini que as primeiras gerações treinariam apenas por cinco épocas, as gerações 3 e 4 correriam por 10 épocas e as restantes por 100 épocas. Esta decisão permitiu encontrar com maior rapidez as melhores combinações. Os passos por época utilizados nesta segunda fase mantiveram-se sempre iguais para que fosse utilizado todo o conjunto de treino. Este valor é calculado através do tamanho do conjunto a dividir pelo *batch_size*.

Poderia também ter opdatado por controlar o número de parâmetros e tentar evitar que fossem produzidos descendentes com uma complexidade computacional elevada através da função de fitness, contudo não me pareceu viável eliminar esses cromossomas estando à procura dos parâmetros ótimos da rede.

Finalmente, foi possível observar os resultados do algoritmo genético ao longo de 8 gerações, com uma população inicial de 8 indivíduos e selecionando os 4

melhores em cada geração.

No gráfico seguinte é possível observar a evolução da *accuracy* da população ao longo das gerações.

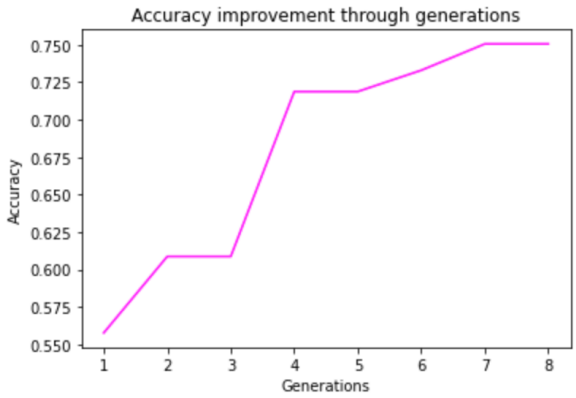


Figura 4.2: Evolução da *accuracy* da população ao longo das gerações

Devido à aleatoriedade da população inicial e de todo o processo, ao longo destas oito gerações foi encontrado o seguinte cromossoma ótimo com um score de 0.75:

- **Número de camadas intermédias:** 2
- **Número de filtros:** 128
- **Função de Ativação:** `selu`
- **Função de Otimização:** `Adagrad`
- **Função de Perda:** `categorical_crossentropy`
- **Taxa de Aprendizagem:** 0.1
- **Padding:** `valid`
- **Dropout:** 0.3

O código desta tentativa pode ser consultado no ficheiro `8gen.ipynb`.

Foram feitas mais algumas tentativas, contudo a que gerou melhores resultados apresenta-se na próxima secção.

4.3 Solução ótima

A solução ótima encontrada através do algoritmo genético com um *score* de 0.84 apresenta-se de seguida:

- **Número de camadas intermédias:** 4

- **Número de filtros:** 256
- **Função de Ativação:** `relu`
- **Função de Otimização:** `RMSprop`
- **Função de Perda:** `categorical_crossentropy`
- **Taxa de Aprendizagem:** 0.001
- **Padding:** `valid`
- **Dropout:** 0.3

Este cromossoma foi encontrado na geração 5 de uma tentativa de correr o algoritmo genético ao longo de 10 gerações. Devido aos limites de utilização das plataformas utilizadas(*Google Colab* e *Kaggle*) não foi possível alcançar o resultado final, no entanto um *score* de 0.84 é bastante satisfatório mesmo sendo obtido num treino cuja *shape* das imagens foi reduzida.

Considero a decisão de não utilizar uma *seed* para permitir que em diferentes experiências houvesse a maior variedade possível tenha sido útil, pois embora em termos de comparação de resultados ao correr mais gerações esta não seria tão fiável, sendo o objetivo obter os parâmetros ótimos este foi alcançado com maior rapidez.

Após a avaliação do modelo com o conjunto de teste foi obtida uma *accuracy* de 80%. Os valores finais do modelo ótimo encontram-se na tabela seguinte. A *shape* das imagens utilizadas para o treino do modelo com os parâmetros ótimos foi a original, isto é, (224,224,3). O *fit* deste modelo foi feito ao longo de 50 épocas. Face ao modelo treinado durante o algoritmo genético com uma *shape* de (100,100,3) a *accuracy* de treino apenas reduziu 4%.

	CNN ótima
Test Loss	0.7926
Test Accuracy	0.8072
Test Precision	0.9324
Test Recall	0.7071

Tabela 4.1: Resultados obtidos após a avaliação do modelo

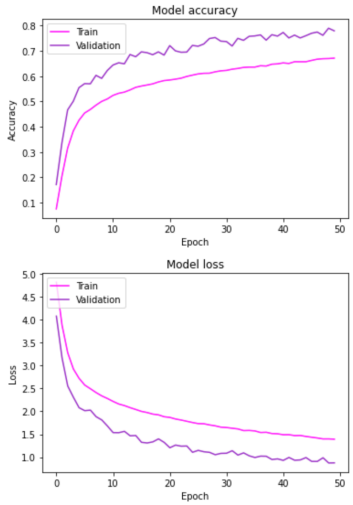


Figura 4.3: Variação da *Accuracy* e *Loss* ao longo das épocas

O código desenvolvido que suporta esta secção pode ser consultado no ficheiro `otimo224.ypynb`, juntamente com o *classification report* no qual está descriminada para as 250 classes a sua *precision* e *recall*. O algoritmo genético que não correu até ao fim onde foi encontrado o resultado ótimo pode ser consultado no ficheiro `10gen.ipynb`.

5 | Transfer Learning

O *transfer learning* é uma técnica de *machine learning* que consiste na aplicação de modelos pré-treinados em determinados conjuntos de dados a outras conjuntos para facilitar a aprendizagem. A aplicação desta técnica é vantajosa na medida em que o treino é realizado com maior rapidez e os resultados obtidos são também melhores. Devido a todo o tempo investido no desenvolvimento do algoritmo genético apenas foram testados modelos de *transfer learning* simples com o intuito de observar as melhorias na *accuracy* e no tempo de treino. Estes modelos foram treinados com as imagens originais, isto é, com uma *shape* (224,224,3).

5.1 VGG16

O modelo `VGG16` é um modelo CNN proposto por K. Simonyan e A. Zisserman no artigo *Very Deep Convolutional Networks for Large-Scale Image Recognition*[3]. Como o o nome indica, o modelo é constituído por 16 camadas.

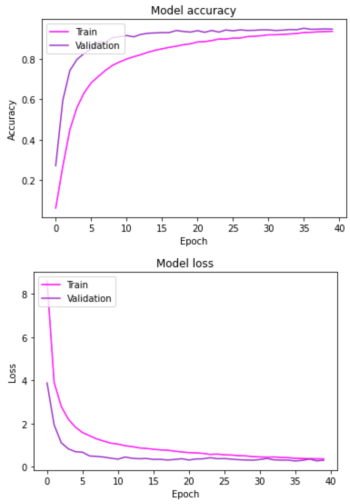
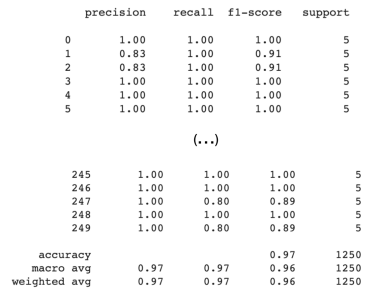


Figura 5.1: *Accuracy* e *Loss* de treino ao longo das épocas

Através deste modelo de *transfer learning* foi obtida uma *accuracy* de 0.9656 com maior eficiência que para os processos descritos anteriormente.



	precision	recall	f1-score	support
0	1.00	1.00	1.00	5
1	0.83	1.00	0.91	5
2	0.83	1.00	0.91	5
3	1.00	1.00	1.00	5
4	1.00	1.00	1.00	5
5	1.00	1.00	1.00	5
		(...)		
245	1.00	1.00	1.00	5
246	1.00	1.00	1.00	5
247	1.00	0.80	0.89	5
248	1.00	1.00	1.00	5
249	1.00	0.80	0.89	5
accuracy			0.97	1250
macro avg	0.97	0.97	0.96	1250
weighted avg	0.97	0.97	0.96	1250

Figura 5.2: Excerto da matriz de classificação deste modelo

```
Predicted a 0. Real value is 0.
Predicted a 0. Real value is 0.
Predicted a 0. Real value is 0.
Predicted a 0. Real value is 0.
Predicted a 0. Real value is 0.
Predicted a 1. Real value is 1.
Predicted a 1. Real value is 1.
Predicted a 1. Real value is 1.
Predicted a 1. Real value is 1.
Predicted a 1. Real value is 1.
Predicted a 2. Real value is 2.
Predicted a 2. Real value is 2.
Predicted a 2. Real value is 2.
Predicted a 2. Real value is 2.
Predicted a 2. Real value is 2.
Predicted a 3. Real value is 3.
Predicted a 3. Real value is 3.
Predicted a 3. Real value is 3.
Predicted a 3. Real value is 3.
Predicted a 3. Real value is 3.
Predicted a 4. Real value is 4.
Predicted a 4. Real value is 4.
Predicted a 4. Real value is 4.
Predicted a 4. Real value is 4.
Predicted a 4. Real value is 4.
```

Figura 5.3: Previsões efetuadas pelo modelo

O código desenvolvido que suporta esta secção pode ser consultado no ficheiro `VGG16.ipynb`.

5.2 VGG19

O modelo `VGG19` é um uma variante da arquitetura `VGG` composto por 19 camadas, 16 dessas camadas são de convolução, 3 *Fully connected*, 5 *MaxPool* e uma camada de *SoftMax*.

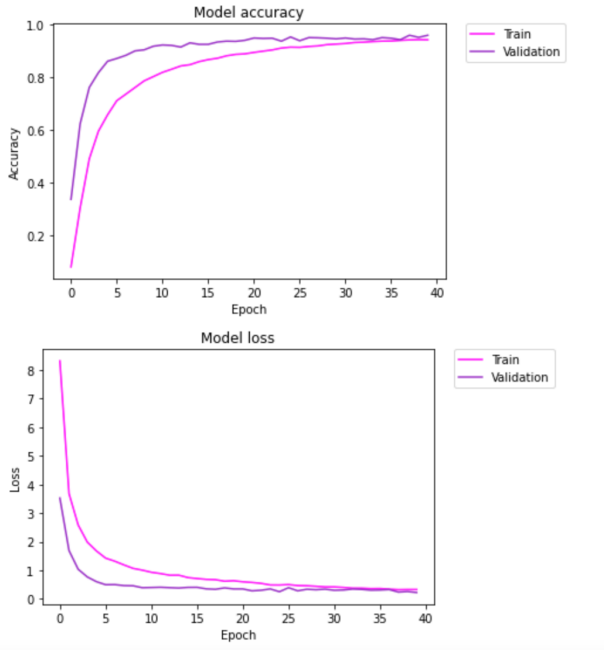


Figura 5.4: *Accuracy* e *Loss* de treino ao longo das épocas

Este modelo obteve uma *accuracy* de 0.9688 para o conjunto de teste.

	precision	recall	f1-score	support
0	1.00	1.00	1.00	5
1	1.00	1.00	1.00	5
2	0.83	1.00	0.91	5
3	1.00	1.00	1.00	5
4	1.00	1.00	1.00	5
5	1.00	1.00	1.00	5
6	1.00	1.00	1.00	5
7	1.00	1.00	1.00	5
	(...)			
245	1.00	1.00	1.00	5
246	1.00	1.00	1.00	5
247	1.00	1.00	1.00	5
248	0.83	1.00	0.91	5
249	1.00	1.00	1.00	5
accuracy			0.97	1250
macro avg	0.97	0.97	0.97	1250
weighted avg	0.97	0.97	0.97	1250

Figura 5.5: Excerto da matriz de classificação deste modelo

O código desenvolvido que suporta esta secção pode ser consultado no ficheiro `VGG19.ipynb`.

6 | Resultados

O desenvolvimento de uma solução ótima foi um processo longo e incremental de tentativa erro. Na tabela 6.1 são apresentados os melhores valores obtidos em cada etapa e é visível o progresso.
Partindo de uma CNN experimental cujos parâmetros foram escolhidos aleatoriamente para efeitos de teste pode observar-se uma *accuracy* para o conjunto de treino quase de 100% o que indica um claro *overfitting* ao longo do treino do modelo pois tanto o conjunto de validação como de teste obteram uma *accuracy* de apenas 64% e 65%, respetivamente. Com a introdução de *data augmentation* verifica-se uma redução na *accuracy* do conjunto de treino, e tal como mencionado em secções anteriores, verifica-se uma redução no *overfitting* dos pesos da rede pois a *accuracy* do conjunto de treino desceu, contudo a de teste e validação aumentou.
Após a aplicação do modelo ótimo encontrado pelo algoritmo genético, a imagens com a *shape* original, observou-se, de novo, uma redução na *accuracy* do conjunto de treino e um incremento na *accuracy* de validação e teste, tendo a ultima chegado aos 80% o que dada a dimensão do conjunto e o número de classes a classificar é um valor bastante satisfatório.
Finalmente, a aplicação de arquiteturas de *transfer learning* permitiu não só agilizar a demorada aprendizagem observada anteriormente, como também obter valores acima de 90% para os três conjuntos e uma classificação de imagens muito mais fiável.

	CNN Inicial	CNN + DA	CNN ótima	VGG16	VGG19
Train Accuracy	0.9973	0.8583	0.6760	0.9363	0.9415
Train Loss	0.0400	0.5240	1.3742	0.3591	0.3264
Validation Accuracy	0.6480	0.7136	0.7792	0.9464	0.9576
Validation Loss	1.4994	1.1335	0.8773	0.3010	0.2273
Test Accuracy	0.6567	0.7544	0.8072	0.9656	0.9688
Test Loss	1.4369	0.9925	0.7926	0.2001	0.2008

Tabela 6.1: Resultados finais

7 | Conclusão

Após a conclusão deste trabalho creio que a construção, classificação e avaliação de uma rede convolucional neuronal, independentemente do número de classes a classificar ficou bastante bem consolidada. Este projeto requereu constante pesquisa, avaliação e tomada de decisões. Esta tomada de decisões foi dificultada em grande parte pelos longos tempos de espera. Foram analisados todos os parâmetros que podem estar presentes na arquitetura deste tipo de redes e selecionados alguns dos mais importantes, mas que não comprometiam a solução a nível computacional, para constituir os cromossomas do algoritmo de otimização. Foram obtidos inúmeros resultados, alguns muito maus, que permitiram perceber as necessidades do conjunto de dados em questão e os erros cometidas na construção de toda a arquitetura, e os melhores que foram relatados neste documento. Estando perante um problema de classificação com 250 classes creio que a arquitetura ótima de uma CNN encontrada com 80% de *accuracy* é bastante satisfatória e creio que em termos de exploração, embora o algoritmo genético tenha trabalhado com uma pequena parte de tudo aquilo que poderia gostaria de ter tido mais tempo e recursos para realmente testar essas opções. Teria sido interessante conseguir explorar no algoritmo genético uma arquitectura cujas camadas fossem mais dinâmicas.

Bibliografia

[1] Gibb, S., La, H. M., Louis, S. (2018, July). A genetic algorithm for convolutional network structure optimization for concrete crack detection. In 2018 IEEE Congress on Evolutionary Computation (CEC) (pp. 1-8). IEEE.

[2] Silaparasetty, N. (2020). The Tensorflow Machine Learning Library. In Machine Learning Concepts with Python and the Jupyter Notebook Environment (pp. 149-171). Apress, Berkeley, CA.

[3] Simonyan, K., Zisserman, A. (2015). Very Deep Convolutional Networks for Large-Scale Image Recognition.