



Universidade do Minho
Escola de Engenharia

Computação Natural

Trabalho Prático Individual 1
MiEI - 4^o Ano - 2^o Semestre

A84167 Susana Vitória Sá Silva Marques

Braga,
8 de abril de 2022

Conteúdo

1	Introdução	2
2	Estado de arte	2
2.1	CNN	2
2.2	Dataset utilizado	3
2.3	Algoritmos genéticos	3
3	Análise dos Dados e Pré-Processamento	3
3.1	Data Augmentation e Balanceamento de Dados	4
4	Modelo CNN	6
4.1	Versão 1	6
4.2	Parâmetros	9
4.3	Versão 2	11
5	Transfer Learning	13
5.1	VGG16	14
5.2	InceptionV3	16
5.3	InceptionResNetV2	17
5.4	Testes e previsões	19
6	Algoritmo genético	20
6.1	Funções de Fitness e de Seleção	21
6.2	Funções de Crossover e de Mutação	21
6.3	Criação do modelo	22
6.4	Resultados	23
7	Conclusão	27

1 Introdução

Neste trabalho, pretende-se desenvolver uma *CNN* e utilizar o conhecimento adquirido em *Genetic Algorithms* para conceber uma solução para o problema de classificação de espécies de aves que aparecem numa imagem.

Assim, pretende-se aplicar conhecimentos ao longo da unidade curricular para a preparação e análise do conjunto de dados proposto, seguido do desenvolvimento e otimização dos modelos de aprendizagem.

2 Estado de arte

2.1 CNN

As CNNs são as redes neuronais que se tornaram na técnica de visão por computador mais moderna e popular. Estes modelos de redes neuronais convolucionais são onnipresentes no mundo de dados de imagens e funcionam fenomenalmente bem em tarefas de visão por computador, como deteção de objetos, reconhecimento e classificação de imagens, o que é exatamente o pretendido neste trabalho ao ter-se como objetivo reconhecer e classificar espécies de aves através de imagens das mesmas.

Através do conhecimento adquirido nas aulas com os *datasets MNIST* e *Fashion MNIST* e depois de uma exploração pelos *datasets CIFAR-10* e *ImageNet* conseguiu-se perceber, de uma maneira prática, o funcionamento de uma *CNN*. Com o *MNIST* e o *Fashion MNIST* é fácil de se ter uma acurácia de 90% ou mais com um simples modelo uma vez que apenas precisamos de classificar 9 classes diferentes (tanto nos números como nas roupas) e as imagens são de 28x28 píxeis.

Algo mais desafiante do que estes *datasets* é o do *CIFAR* que consiste em 60 mil imagens 32x32, com cores e com 10 classes, tendo 6 mil imagens por classe. Existe 50 mil imagens para treino e 10 mil imagens para teste. Pontos importantes a distinguir deste *dataset* para o do *MNIST* são as cores em comparação com a escala de cinzento do último, o tamanho de cada imagem e a quantidade de dados. [1]

Tal como no *dataset* que nos foi fornecido estas imagens têm condições de luz variadas e ângulos diferentes, uma arquitetura CNN simples, como a do *MNIST* iria resultar numa acurácia por volta dos 60%. Numa breve pesquisa, encontraram-se resultados mais satisfatórios (entre 78% a 80%) ao alterar o modelo: ao aumentar o número de camadas *Conv2D* para criar um modelo mais profundo, ao aumentar números de filtros para aprender mais *features*, ao adicionar *Dropout* e mais *Dense layers* de forma a regularizar.

Ao se progredir com a pesquisa inicial, o *dataset Imagenet* é o próximo passo para aumento de complexidade. Como breve referência, este *dataset* é considerado as Olimpíadas de Visão por Computador visto que todos os anos equipas investigadoras e académicas tentam competir com novos e melhores algoritmos em tarefas de Visão por Computador com este *dataset*. [1]

Ao contrário dos casos anteriores discutidos, as imagens no *ImageNet* têm uma resolução decente (224x224) e processar um *dataset* deste tamanho requer um poder computacional muito maior em termos de CPU, GPU e RAM (tal como acontece no *dataset* que nos é proposto trabalhar. Neste caso o uso de mode-

los CNN, mesmo mais complexos garantem uma acurácia de 40-50%, assim é recomendado o uso de *transfer learning*, que iremos falar posteriormente, para melhorar a acurácia.

2.2 Dataset utilizado

Depois de grandes pesquisas sobre qual seria a acurácia que o modelo deveria obter para ser considerado um bom modelo, entendemos que sem *transfer learning* por volta de 60% é aceitável num *dataset* desta capacidade.

Utilizar *fine tuning* e um algoritmo genético é uma mais valia para conseguir bons resultados, mas durante a pesquisa, verificou-se que com *transfer learning*, principalmente usando o *dataset InceptionResNet* encontrava-se uma melhor acurácia a rondar os 90%. Porém enquanto se realizava este trabalho usando vários métodos diferentes, percebeu-se que todos estes *datasets* que foram pesquisados e encontrados como forma de avaliação dos resultados obtidos eram apenas parecidos com este *dataset*, mas nunca iguais, sendo o *dataset* fornecido desbalanceado e numa fase inicial gerar um modelo muito *overfitted* se nada ao contrário se fizer para o mudar devido principalmente a existir muitos dados de treino do que de validação ou teste. Desta forma os resultados obtidos, nunca serão tão bons como os encontrados com 90% uma vez que para além do problema acima descrito, ainda existe uma escassez de poder computacional a nível de GPU e de recursos como por exemplo o tempo que o modelo demora a correr usando o *dataset* em causa

2.3 Algoritmos genéticos

Algoritmos genéticos são algoritmos que possibilitam um processo automático de otimização de modelos de *machine learning*, com um baixo custo de implementação, tendo apenas como defeito a quantidade de tempo necessária para treinar. Dada a possibilidade de tantas arquiteturas CNN possíveis, como se viu anteriormente, escolher uma é dispendioso e não praticável, sendo que se se focar numa ferramenta por trás como o **Algoritmo Genético** este processo torna-se mais simples. Neste trabalho, usa-se um Algoritmo Genético totalmente implementado para otimizar os valores procurados de acurácia e de *loss* para o problema que nos é proposto, usando o melhor modelo CNN obtido. [3]

3 Análise dos Dados e Pré-Processamento

Este dataset que nos foi entregue num ficheiro *.zip* é referente a um conjunto de imagens de diversas aves com intuito a se conseguir fazer a sua classificação. Ele é composto por 3 diretorias: *train*, *valid* e *test*, que correspondem à divisão correta de forma a se treinar o modelo pretendido, ou seja treinou-se o modelo com as imagens que estão na diretoria *train*, depois validamos com as que estão na diretoria *valid* e por fim testamos com as que estão na diretoria *test*.

Seguindo o enunciado e sabendo que o projeto seria feito usando recursos de CPU e GPU bastante elevados, desde o início decidiu-se usar o *Google Collab* para uma maior eficiência a nível de *performance*, apesar deste na sua versão gratuita também ter algumas limitações: apenas é possível correr durante 12 horas consecutivas e se se usar GPU, no final de um máximo de 6 horas, muitas

vezes menos, a máquina da *Google* é nos retirada com o tempo de execução reiniciado e eliminando toda a RAM disponível. Este facto fez com que inúmeras vezes depois de correr algoritmos por 3 ou 4 horas o trabalho dos mesmos fosse completamente eliminado, sem se ter a opção de guardar os resultados, apenas o output do *notebook* (mesmo gravando o ficheiro *.h5*).

Este *dataset* contém para treino 35215 imagens que pertencem a 250 classes distintas, 1250 imagens para validação e 1250 imagens para teste que pertencem igualmente às mesmas 250 classes encontrando-se inicialmente desbalanceado. Tal e qual como os *datasets* mais complexos referidos no estado de arte, este é constituído por imagens coloridas complexas e de diversos ângulos de 250 tipos de aves diferentes com uma resolução de 224x224.

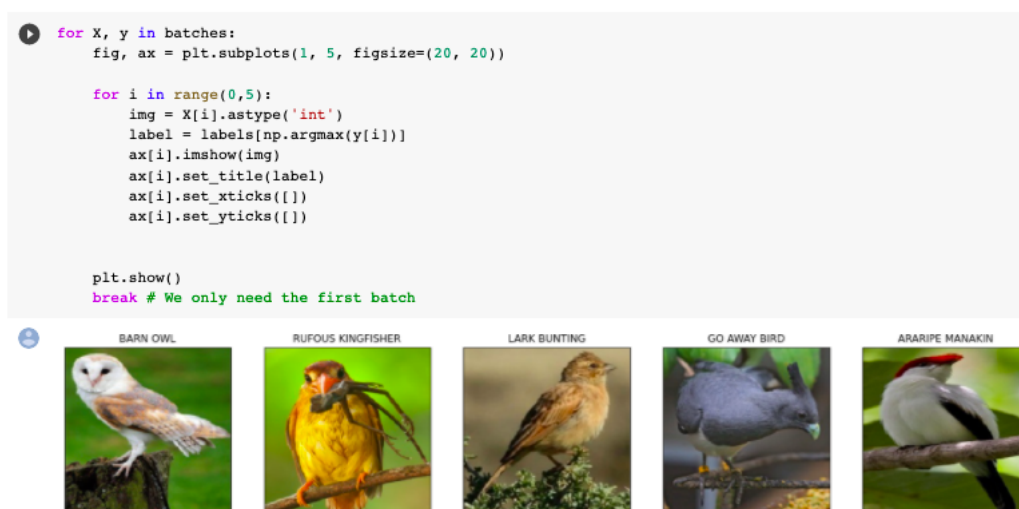


Figura 1: Visualização de imagens de treino

3.1 Data Augmentation e Balanceamento de Dados

Para um carregamento de dados mais rápido e dinâmico a biblioteca *Keras* tem uma funcionalidade bastante interessante que não maximiza a RAM ao usar *batches* pequenos.

Assim usando *ImageDataGenerator* e *flow_from_directory* que deduz as imagens e *labels* para cada classe e o número de classes através de cada diretoria, as imagens de todas as diretorias (*train*, *valid* e *test*) foram inicialmente carregadas com um *batch size* de 256 uma vez que existe uma grande quantidade de dados, com o *color mode* igual a *rgb* já que são imagens com cor ao invés de apenas preto e branco, e o seu modo é categórico, pois pretende-se classificar categoricamente 250 classes diferentes.

É necessário ainda fazer um *rescale* para $1/255$ para transformar todos os pixéis com um valor entre 0 a 255 para um valor entre 0 a 1, para se poder tratar todas as imagens da mesma maneira (assim como todas as imagens usam o mesmo modelo, pesos e *learning rate*, imagens com pixéis com intervalos maiores tendem a criar uma *loss* maior e a soma destas vai contribuir para a atualização do modelo mais tarde. Assim este *rescale* ajuda bastante na progressão do *feature*

learning.

Para além deste tratamento, ainda se usou *data augmentation*, que é uma estratégia que permite aumentar a diversidade de dados disponíveis para os modelos de treino sem realmente colecionar novos dados. Para isso usou-se diferentes técnicas com a ajuda do *ImageDataGenerator* como o *horizontal flip* e o *vertical flip* que permitem criar imagens com reflexão, o *zoom range* que cria novas imagens ampliadas, o *rotation range* que permite criar imagens com um certo grau de rotação, o *shear range* que permite fazer um *shift* às imagens e o *fill mode* com o campo *nearest* que permite preencher, por exemplo depois de uma rotação, as novas áreas da imagem com os píxeis mais próximos descobertos. Devido à qualidade de imagens ser bastante elevada (224x224) o tempo gasto para o modelo as processar é muito superior ao disponível para este trabalho se se pretender fazer diversos testes com diversos modelos e não apenas correr apenas um. Por exemplo, mesmo usando GPU usando um *target size* = (224,224), cada *epoch* iria demorar cerca de 8 minutos a correr. Querendo aproveitar o máximo de tempo disponível para fazer vários testes, usar diversos modelos de *transfer learning* e finalmente fazer otimizações de parâmetros com o algoritmo genético, decidiu-se reduzir a qualidade de imagens para um número que se pensa não afetar demasiado a aprendizagem do modelo. Como as imagens tinham 224x224, dividindo este número por 4 para apenas escalar a imagem sem reduzir o seu formato em apenas uma das duas componentes (largura e altura) optou-se por ter um *target size* = (56,56) em todos os modelos com exceção do modelo genético e do modelo com *transfer learning InceptionResNet* uma vez que para este modelo só é possível utilizar imagens com *target size* de 75 píxeis por 75 píxeis ou mais.

```
#Data Augmentation
train_datagen = ImageDataGenerator(rescale = 1/255,
                                   horizontal_flip=True,
                                   vertical_flip=True,
                                   zoom_range=0.2,
                                   shear_range=0.2,
                                   rotation_range=30,
                                   fill_mode='nearest')

valid_datagen = ImageDataGenerator(rescale = 1/255)

test_datagen = ImageDataGenerator(rescale = 1/255)
```

Finalmente, como o enunciado dizia e se verificou após alguns testes, os dados de treino fornecidos encontravam-se desbalanceados, existindo para cada classe diferentes números totais de imagens. Para os balancear criou-se uma função com um dicionário de *python* que permite através da biblioteca *numpy* balancear os mesmos ao associar os pesos às respetivas *labels* e depois adicionar essa função ao *model.fit* alimentando esses pesos como parâmetros para os pesos da classe (*class_weight*) como se pode ver no segmento de código a seguir:

```
class_weights = dict(zip(np.unique(train_generator.classes),
                          class_weight.compute_class_weight('balanced',
```

```

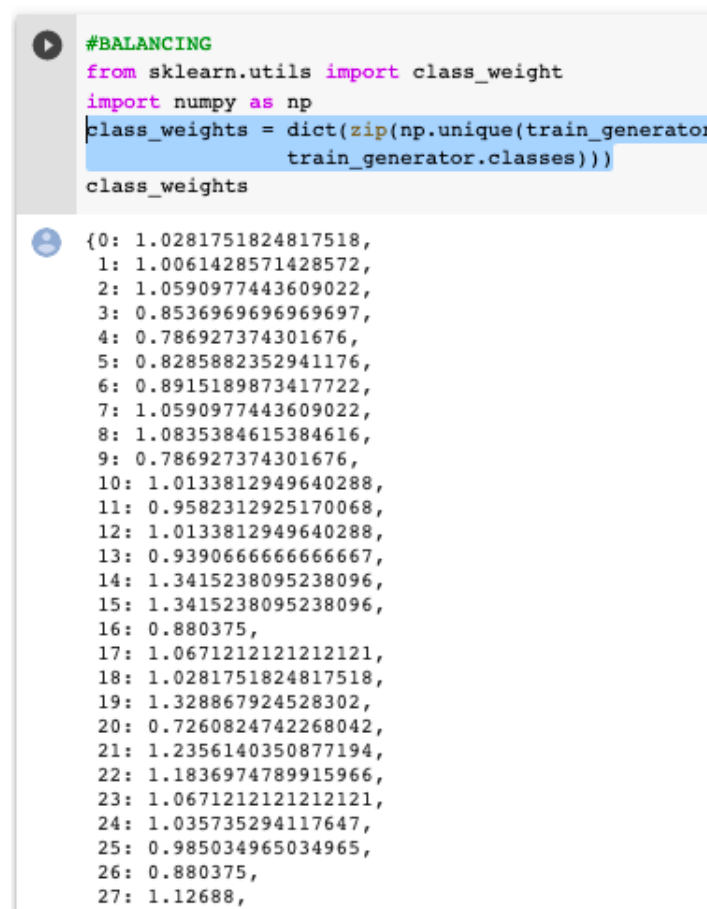
        np.unique(train_generator.classes),
        train_generator.classes)))

...

history = model.fit(train,...,
                    validation_data=val, class_weight=class_weights)

```

A figura seguinte mostra um excerto do resultado da função criada (`class_weights`) e como altera estes pesos para cada classe, verificando que classes com menos imagens (como por exemplo a classe 0) têm pesos maiores que classes com mais imagens (como por exemplo a classe 4), balanceando assim o *dataset*.



```

#BALANCING
from sklearn.utils import class_weight
import numpy as np
class_weights = dict(zip(np.unique(train_generator
                                   train_generator.classes)))
class_weights

```

```

{0: 1.0281751824817518,
 1: 1.0061428571428572,
 2: 1.0590977443609022,
 3: 0.8536969696969697,
 4: 0.786927374301676,
 5: 0.8285882352941176,
 6: 0.8915189873417722,
 7: 1.0590977443609022,
 8: 1.0835384615384616,
 9: 0.786927374301676,
10: 1.0133812949640288,
11: 0.9582312925170068,
12: 1.0133812949640288,
13: 0.9390666666666667,
14: 1.3415238095238096,
15: 1.3415238095238096,
16: 0.880375,
17: 1.0671212121212121,
18: 1.0281751824817518,
19: 1.328867924528302,
20: 0.7260824742268042,
21: 1.2356140350877194,
22: 1.1836974789915966,
23: 1.0671212121212121,
24: 1.035735294117647,
25: 0.985034965034965,
26: 0.880375,
27: 1.12688,
...

```

Figura 2: Pesos respetivos de cada classe após balanceamento

4 Modelo CNN

4.1 Versão 1

Numa primeira análise e construção do modelo, existindo uma pouca aprendizagem sobre o número de camadas a colocar e o número de parâmetros e

escolha dos mesmos, criou-se um modelo que apesar de se encontrar com uma acurácia final razoável, apresentava bastante *overfitting*, uma vez que a *val_loss* encontrava-se sempre muito acima da *train_loss* apesar de descer. Mesmo o decréscimo da *val_loss* era muito instável, tendo que correr o modelo diversas vezes guardando os resultados em memória para conseguir chegar-se a uma acurácia de 52%.

Modelo:

```
model = Sequential()
#1. LAYER
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=(56, 56, 3)))
model.add(MaxPool2D(pool_size=(2, 2)))
model.add(BatchNormalization())
model.add(Dense(100))
model.add(Activation("relu"))

#2. LAYER
model.add(Conv2D(filters = 32, kernel_size = (3,3), padding = 'Same'))
model.add(MaxPool2D(pool_size=(2, 2)))
model.add(BatchNormalization())
model.add(Dense(200))
model.add(Activation("relu"))

#3. LAYER
model.add(Conv2D(filters = 32, kernel_size = (3,3), padding = 'Same'))
model.add(MaxPool2D(pool_size=(2, 2)))
model.add(BatchNormalization())
model.add(Dense(100))
model.add(Activation("relu"))

#4. LAYER
model.add(Conv2D(filters = 32, kernel_size = (3,3), padding = 'Same'))
model.add(MaxPool2D(pool_size=(2, 2)))
model.add(BatchNormalization())
model.add(Dense(150))
model.add(Activation("relu"))

#5. LAYER
model.add(Conv2D(filters = 64, kernel_size = (3,3), padding = 'Same'))
model.add(BatchNormalization())
model.add(Activation("relu"))

#5. LAYER
model.add(Conv2D(filters = 64, kernel_size = (3,3), padding = 'Same'))
model.add(BatchNormalization())
model.add(Dropout(0.1))
```



```

model.add(Activation("relu"))

# Flatten the results to one dimension for passing into our final layer
model.add(Flatten())
# A hidden layer to learn with
model.add(BatchNormalization())
model.add(Dense(560, activation='relu'))
# Another dropout
model.add(Dropout(0.15))
# Final categorization from 0-9 with softmax
# output layer
model.add(Dense(250, activation='softmax'))
model.summary()
optimizer = Adam(lr=0.001, beta_1=0.9, beta_2=0.999,
                  epsilon=None, decay=0.0, amsgrad=False)
# compile model
model.compile(loss='categorical_crossentropy',
              optimizer=optimizer, metrics=['accuracy'])

```

Resultados:

Train Results

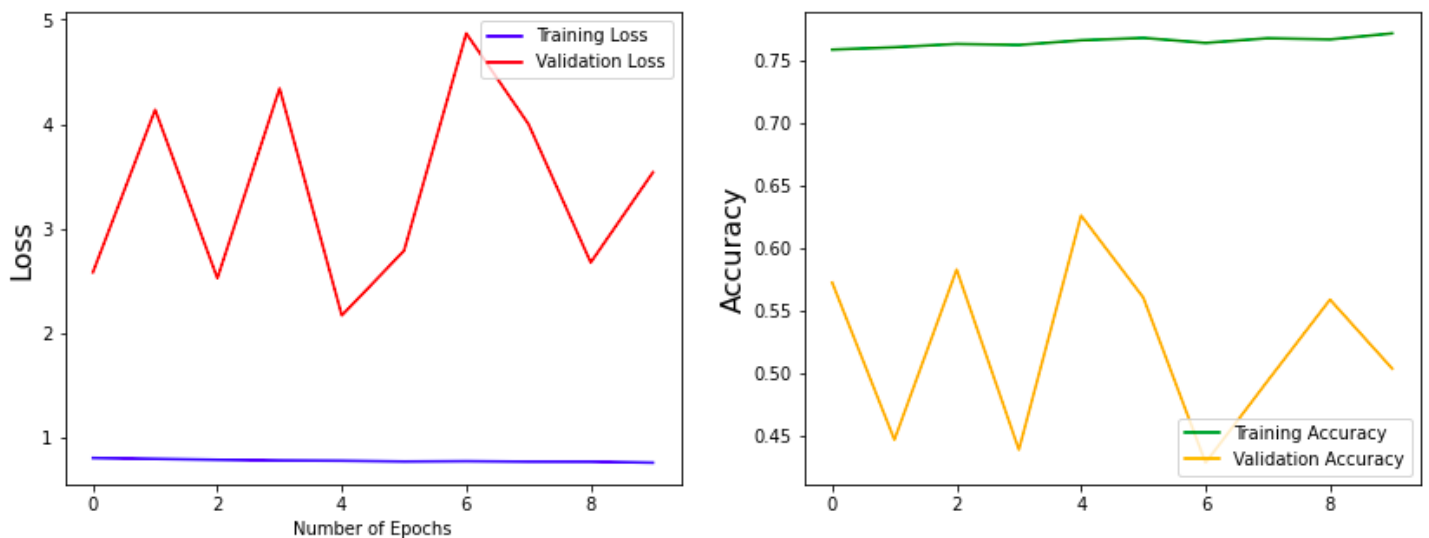


Figura 3: Variação da acurácia e da *loss* do primeiro modelo criado (overffited)

Train accuracy of the model: 0.7712054252624512

Train loss of the model: 0.7646011114120483

Validation accuracy of the model: 0.5031999945640564

Validation loss of the model: 3.540607452392578

Test Loss: 3.2961008548736572

Test Accuracy: 0.5343999862670898

Como se pode observar pelo gráfico o modelo está completamente *overfitted* com os resultados da *validation loss* e *validation accuracy* muito flexíveis aumentando e diminuindo ao longo do tempo, fazendo com que apesar do modelo tenha uma acurácia de 0.5, seja um modelo muito pouco eficaz e não o pretendido para o trabalho.

Compreendeu-se depois de uma pesquisa [3] que ao usar tantas camadas e com tão poucos *Dropouts* e *BatchNormalizations* que o modelo se encontrava *overfitted* devido a uma má conjugação destes números. Assim o próximo passo foi realizar uma maior investigação sobre quais os melhores parâmetros a usar e fazer um pouco de *fine-tuning* para gerar um modelo mais avançado e que não esteja *overfitted*. Com este objetivo, na próxima secção faz-se um resumo dos parâmetros que podemos e devemos alterar e o que realmente cada um deles faz, para se poder atingir um melhor resultado.

4.2 Parâmetros

- Função de ativação: O objetivo da função de ativação é introduzir não linearidade na saída de um neurónio que faz com que este possa aprender mais do que se tivesse uma relação linear entre as variáveis dependentes e independentes. Por outro lado, com esse poder a mais surgem algumas dificuldades. Particularmente, ao introduzir uma ativação não linear, a superfície de custo da rede neuronal deixa de ser convexa, tornando a otimização mais complicada. Além disso, algumas não linearidades tornam o problema dos gradientes crescente ou diminuindo mais evidente. A ativação ReLu usa uma função com o mesmo nome que é fácil de otimizar (já que é parecida com a função identidade) e produz zeros em metade do seu domínio. Assim ela é muito eficiente e a sua não linearidade é um ótimo exemplo de como a simplicidade pode ser extremamente poderosa. Uma desvantagem é que durante o treino é possível o neurónio começar a produzir apenas zeros. Isto acontece quando a soma ponderada antes da aplicação da ReLu se torna negativa. Nessa região, a derivada também é zero, fazendo com que os parâmetros deixem de ser atualizados com o gradiente descendente. Para solucionar o problema da ReLu, uma proposta é dar uma inclinação *alpha* para a função na parte negativa do seu domínio, surgindo assim a função Leaky ReLu que é bastante parecida com a identidade e tem as derivadas estáveis. A ativação ELU, tal como a LeakyReLu resolve os problema apresentado pelas ReLUs e a não linearidade desta também é concentrada na parte negativa do seu domínio. Uma desvantagem de usar ELUs é que a função exponencial é bastante ineficiente, tornando redes neuronais com ELUs mais lentas.
- Função de loss: A função de perda é um método de avaliar quão bem o modelo lida com o conjunto de dados. Caso o modelo estiver mal treinado, o que acontece normalmente em função dos dados utilizados, a função de perda produzirá um valor elevado. Se o modelo for muito bom, o resultado será um número menor. A medida que se altera partes do algoritmo para tentar aprimorar o modelo, a função de perda informa se se está chegando a algum lugar. Neste caso vamos utilizar a Categorical Cross-entropy, dado que se está a abordar um problema de classificação categórica (250 classes diferentes para prever).

- **Função de otimização:** Com uma função de *loss* definida, precisa-se então ajustar os parâmetros de forma a que a *loss* seja reduzida. CNNs possuem muitos parâmetros que precisam ser aprendidos fazendo com que seja necessário treiná-los usando milhares, ou até milhões, de imagens. Entretanto, realizar a otimização usando milhões de instâncias torna a utilização do Gradiente Descendente (padrão) inviável, uma vez que esse algoritmo calcula o gradiente para todas as instâncias individualmente. Algumas alternativas são SGD, AdaGrad, RMSProp e Adam. Gradiente Descendente Estocástico (do inglês Stochastic Gradient Descent, SGD) utiliza métodos que oferecem aproximações do Gradiente Descendente, por exemplo usando amostras aleatórias dos dados ao invés de analisando todas as instâncias existentes. Por esse motivo, o nome desse método é Gradiente Descendente Estocástico, já que ao invés de analisar todos os dados disponíveis, analisa-se apenas uma amostra, e dessa forma, adiciona-se aleatoriedade ao processo. O AdaGrad busca dar mais importância a parâmetros pouco utilizados. Isso é feito mantendo um histórico de quanto cada parâmetro influenciou a *loss*, acumulando os gradientes de forma individual. Essa informação é então utilizada para normalizar o passo dado em cada parâmetro. Como o gradiente é calculado com base no histórico e para cada parâmetro de forma individual, parâmetros pouco utilizados terão maior influência no próximo passo a ser dado. RMSProp calcula médias da magnitude dos gradientes mais recentes para cada parâmetro e usa-as para modificar o *learning rate* individualmente antes de aplicar os gradientes. Esse método é similar ao AdaGrad, porém, nele a acumulação de gradientes é calculada usando uma média com decaimento exponencial e não com a simples soma dos gradientes. Adam utiliza uma ideia similar ao AdaGrad e ao RMSProp, apenas alterando a sua fórmula matemática usando o *momentum* para calcular o momento de primeira ordem e segunda ordem.
- **Batch Size:** Este parâmetro informa a rede neuronal sobre o tamanho de cada porção de dados que se pretende aprender de cada vez, uma vez que é boa prática não passar o conjunto de dados inteiro para a rede de uma só vez. O valor padrão é 32, podendo subir-se e descendo (tendo em conta que os valores são definidos normalmente em potência de 2) conforme queremos aumentar ou não o tempo do modelo a convergir. Se o *batch size* for menor que 32, o modelo vai demorar mais tempo a aprender e irá ter maior ruído podendo aprender de forma errada mais facilmente por ter menos dados e encontrar ambiguidades e enganos. Mas também pode aprender com este ruído mais rapidamente, por isso é uma questão de testar diferentes números e de gerir os recursos que se tem, nomeadamente tempo.
- **Camada de input:** Esta camada manterá os valores de píxeis dados para a imagem.
- **Camada CONV:** Esta camada calculará a saída de neurônios conectados às regiões locais na entrada, cada um calculando um produto escalar entre os seus pesos e uma região à qual eles estão conectados na entrada.
- **Filtros/Kernel:** A profundidade da saída de uma convolução é igual à quantidade de filtros aplicados. O filtro, também conhecido por Kernel, é for-

mado por pesos inicializados aleatoriamente, atualizando-os a cada nova entrada durante o processo de *backpropagation*. Além do tamanho do filtro e o stride da convolução como hiperparâmetro para extrair características, uma CNN também precisa de *padding*. O *padding* pode não existir, no qual o output da convolução ficará no seu tamanho original, ou existir com 0 (zero *padding*) onde uma borda é adicionada e preenchida com 0's. O *padding* serve para que as camadas não diminuam muito mais rápido do que é necessário para a aprendizagem.

- Camada de pooling: Os pooling's são necessários para reduzir a quantidade de características por filtro (redução de escala). Assim esta camada realizará uma operação de *downsampling* ao longo das dimensões espaciais (costuma ser (2,2) e em casos raros (3,3)). O que ela faz, caso seja 2x2 é selecionar uma matriz com 4 píxeis da imagem e reduzi-la para o valor maior destes píxeis (se se tiver a falar de *MaxPooling*) ou para o valor médio destes píxeis (se se tiver a falar de *AvgPooling*). Se for 3x3, a matriz terá 9 píxeis. Repete-se este processo para toda a imagem selecionando a matriz da esquerda para a direita e de cima para baixo continuamente até chegar ao final.
- Camada FLATTEN: O papel desta camada é realizar uma operação para nivelar a saída da camada anterior, de forma a que as suas dimensões possuam a mesma forma da camada seguinte.
- Dropout: É uma técnica que previne a co-adaptação dos neurónios, ou seja quer-se estimular unidades a aprender a extrair e representar características dos dados de entrada sem depender de seus vizinhos e produzir uma representação excessivamente distribuída (*overfitting*). Para isso seleciona-se aleatoriamente conjuntos diferentes de neurónios e força-se a ter uma saída nula para diferentes dados de entrada. Na prática isso retira os neurónios da rede nessa iteração de treino.
- Batch normalization: É uma técnica de normalização(pré-processamento usada para *padronizar* os dados) feita entre as camadas de uma rede neuronal em vez de nos dados concretos. É realizada em pequenas porções de dados (batches) em vez do conjunto completo e serve para agilizar o treino e utilizar *learning rate* mais elevado, facilitando a aprendizagem.
- Learning Rate: É um hiperparâmetro que controla o quanto se deve alterar no modelo em resposta ao erro estimado de cada vez que os pesos do modelo são atualizados. Costuma ter um alcance entre 0.0 e 1.0. Escolher o *learning rate* é desafiador, pois um valor pequeno pode resultar num processo longo de treino e custoso, enquanto que um valor muito grande pode resultar na aprendizagem de um conjunto sub-ótimo de pesos muito rápido ou um processo de treino instável. Assim o *learning rate* controla o quão rápido o modelo se adapta ao problema.

4.3 Versão 2

A versão 2 do modelo CNN deste trabalho, já foi mais pensada antes de se realizar com o intuito de melhorar os resultados e principalmente não obter um modelo *overfitted*.

Assim, esta versão bastante melhorada encontra-se a seguir:

```
model = Sequential()
#1. LAYER
model.add(Conv2D(16, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=(56, 56, 3)))
model.add(MaxPool2D(pool_size=(2, 2)))
model.add(Conv2D(filters = 32, kernel_size = (3,3), padding = 'Same'))
model.add(Dense(550))
model.add(Activation("relu"))
model.add(Dropout(0.2))

#2. LAYER
model.add(Conv2D(filters = 32, kernel_size = (3,3), padding = 'Same'))
model.add(MaxPool2D(pool_size=(2, 2)))
model.add(Conv2D(filters = 64, kernel_size = (3,3), padding = 'Same'))
model.add(Dense(550))
model.add(Activation("relu"))
model.add(Dropout(0.2))

# Flatten the results to one dimension for passing into our final layer
model.add(Flatten())
# A hidden layer to learn with
model.add(BatchNormalization())
model.add(Dense(100, activation='relu'))
# Another dropout
model.add(Dropout(0.25))
# Final categorization from 0-9 with softmax
# output layer
model.add(Dense(250, activation='softmax'))
model.summary()
optimizer = Adam(lr=0.001, beta_1=0.9, beta_2=0.999,
epsilon=None, decay=0.0, amsgrad=False)

# compile model
model.compile(loss='categorical_crossentropy', optimizer=optimizer,
metrics=['accuracy'])
```

Como se pode observar, esta versão já se encontra com muito menos camadas (apenas 3) e com mais *Dropouts* e *Batch Normalization*, gerando um modelo que não se encontra em *overfitting* e onde a taxa de validação tanto na *loss* como na acurácia é bastante melhor em ambos os casos (sendo muito menor do que na *train loss* e maior do que a *train accuracy*). Também estas taxas são muito mais estáveis com a acurácia da *validation* sempre a subir e a *loss* a descer.

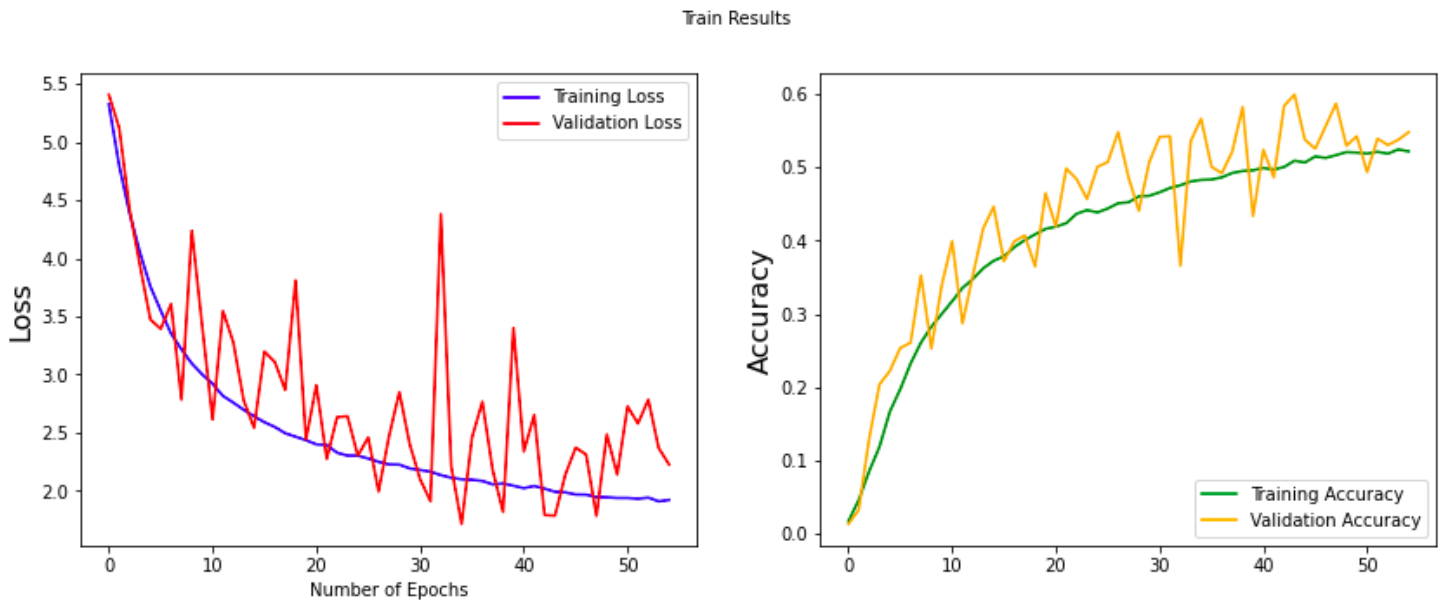


Figura 4: Variação da acurácia e da *loss* do modelo CNN sem overfitting

Train accuracy of the model: 0.5217946767807007

Train loss of the model: 1.9213305711746216

Validation accuracy of the model: 0.5479999780654907

Validation loss of the model: 2.2256269454956055

Test Loss: 2.1052865982055664

Test Accuracy: 0.5856000185012817

Se se utilizasse mais recursos e se tivesse mais tempo, era muito provável que se conseguiria atingir resultados melhores com um maior número de *epochs* e de paciência na *val.loss*.

Mesmo assim, os resultados foram bastante satisfatórios obtendo uma acurácia a rondar os 60 %, dando resultados parecidos com os das pesquisas anteriormente feitas com modelos CNN simples.

5 Transfer Learning

Apesar da acurácia na última secção ter dado resultados satisfatórios, pretende-se ir um pouco mais afundo na questão de treinar estes modelos e obter melhores resultados.

Modelos CNN podem demorar dias ou até semanas para treinar grandes *datasets*. Uma maneira para encurtar este processo é reutilizar um modelo já pré-treinado como já falamos anteriormente. Modelos com uma *performance* de topo podem ser descarregados e usados diretamente ou integrados num novo modelo para atingir uma acurácia superior.

Com *transfer learning* usa-se modelos treindados num problema como um ponto inicial para a resolução de um problema relacionado. É flexível, permitindo que se faça *feature extraction* diretamente no pré-processamento e se integre num novo modelo o que se aprendeu. *Keras* concede acesso a vários modelos de topo como o *VGG16*, o *InceptionV3* e o *InceptionResNetV2* usados neste trabalho. *Transfer learning* tem o benefício de diminuir o tempo de treino de um modelo e pode resultar numa diminuição generalizada de erro.

Os pesos usados em camadas usadas podem ser usados como um ponto inicial para o processo de treino e adaptados em resposta ao novo problema. Isto ajuda bastante quando o primeiro problema tem muitos mais dados que o problema de interesse e a similaridade na estrutura do problema pode ser usada nos dois contextos. [2]

A seguir encontram-se os melhores modelos pré-treinados e a sua acurácia:

Model	Year	Number of Parameters	Top-1 Accuracy
VGG-16	2014	138 Million	74.5%
ResNet-50	2015	25 Million	77.15%
Inception V3	2015	24 Million	78.8%

Figura 5: Modelos pré-treinados

Neste trabalho optarse-á por usar o *VGG16*, o *InceptionV3*, mas ao contrário do que a tabela apresenta, o último modelo a ser estudado será o *InceptionResNetV2*, que é um modelo superior ao *ResNet-50* a nível de acurácia:

Model	Size	Top-1 Accuracy	Top-5 Accuracy
VGG16	528MB	0.715	0.901
ResNet50	99MB	0.759	0.929
InceptionV3	92MB	0.788	0.944
Xception	88MB	0.790	0.945
InceptionResNetV2	215MB	0.804	0.953

Figura 6: Modelos pré-treinados

5.1 VGG16

VGG16 é uma arquitetura CNN que ficou em primeiro lugar no *ImageNet Challenge* em 2014. Ele tem 16 camadas no total com 13 camadas convolucionais. Descarregou-se os pesos deste modelo para as suas propriedades serem utilizadas na tarefa proposta.

```
pretrained_model = VGG16(include_top=False,
                          input_shape=(56,56,3), weights='imagenet')
```

Congelou-se as camadas que se usou no modelo pré-treinado uma vez que as imagens do *imagenet* são comparáveis com as imagens do *dataset* que se está a explorar, assim poupou-se bastante tempo computacional e permitiu melhores tempos de treino, apesar da acurácia reduzir levemente. Ao congelar as camadas, o que se fez foi impedir a propagação de volta às camadas anteriores do modelo, assim os pesos não são alterados e a informação aprendida não é perdida.

Este congelamento foi feito em todos os modelos de *transfer learning* usados no trabalho uma vez que todos eles usam imagens do *Imagenet dataset*.

```
for layer in pretrained_model.layers:
    layer.trainable = False
pretrained_model.summary()
```

```
block5_pool (MaxPooling2D) (None, 1, 1, 512) 0
=====
Total params: 14,714,688
Trainable params: 0
Non-trainable params: 14,714,688
```

Figura 7: Parâmetros não treinados do VGG16 devido ao congelamento

Como modelo final, adicionou-se ao modelo pré-treinado o seguinte modelo:

```
x = keras.layers.Flatten()(pretrained_model.output)

# Add a fully connected layer with 4096 hidden units and ReLU activation
x =keras.layers.Dense(4096, activation='relu')(x)

# Add a dropout rate of 0.20
x =keras.layers.Dropout(0.20)(x)

# Add a final softmax layer for classification
x = tf.keras.layers.Dense(250, activation='softmax')(x)

# compile the model
model2 = tf.keras.models.Model(pretrained_model.input,x)
optimizer = Adam(lr=0.001, beta_1=0.9,
beta_2=0.999, epsilon=None, decay=0.0, amsgrad=False)
model2.compile(loss='categorical_crossentropy',
optimizer=optimizer, metrics=['accuracy'])
```

Com este modelo final, os resultados foram melhores que o modelo da versão 1, sem *overffited*, mas a sua acurácia foi bastante baixa, como podemos verificar:

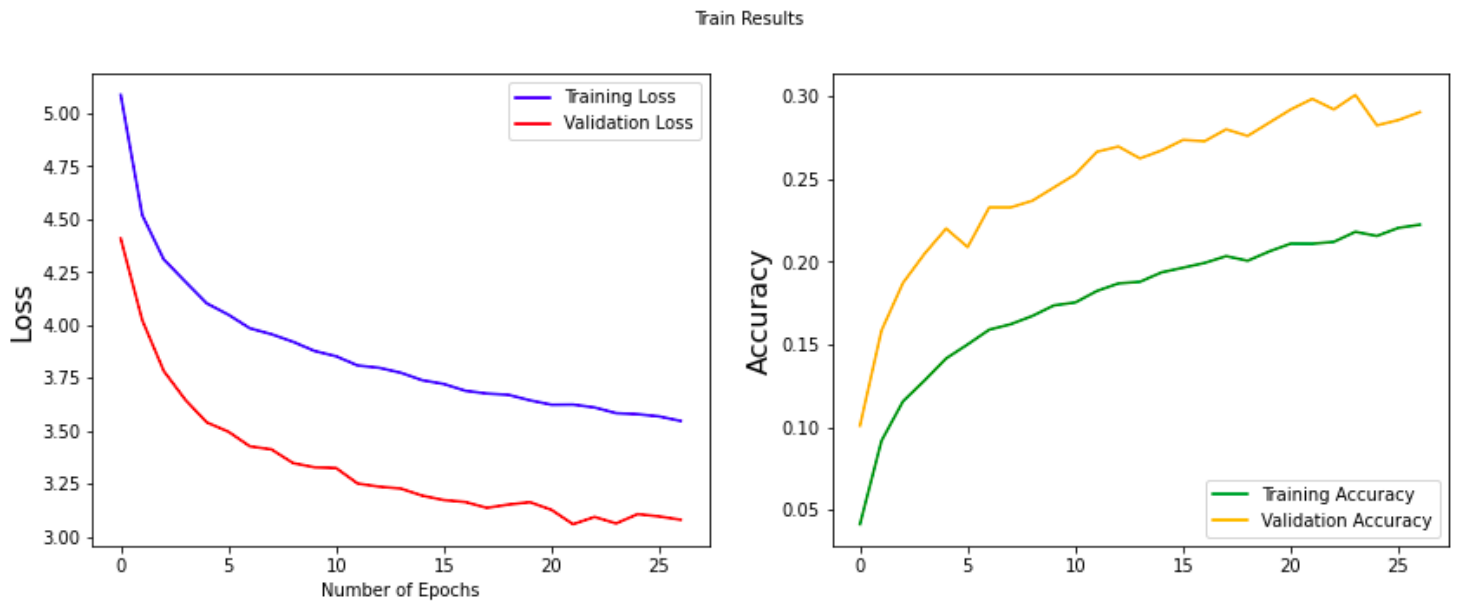


Figura 8: Variação da acurácia e da *loss* do modelo com transfer-learning VGG16

Train accuracy of the model: 0.2223484367132187

Train loss of the model: 3.546396017074585

Validation accuracy of the model: 0.2903999984264374

Validation loss of the model: 3.0793635845184326

Test Loss: 2.964756965637207

Test Accuracy: 0.30239999294281006

Estes resultados mais baixos podem ter sido obtidos pelo número de parâmetros no modelo pré-treinado ser bastante grande em comparação com o número de dados do nosso *dataset* (138 milhões). Assim, concluiu-se que o modelo VGG16 não obteve uma *performance* satisfatória.

5.2 InceptionV3

InceptionV3 é uma terceira iteração da arquitetura *inception* desenvolvida pelo modelo *GoogLeNet*. Este modelo tem 48 camadas e uma acurácia de 78,8 %. Descarregando o modelo pré-treinado com o *Keras* e utilizando os mesmos métodos e parâmetros que no modelo *VGG16*, este modelo encontra-se ligeiramente acima a níveis de acurácia em comparação com o *VGG16* mas ainda é bastante insatisfatório comparando com o modelo CNN- versão 2.

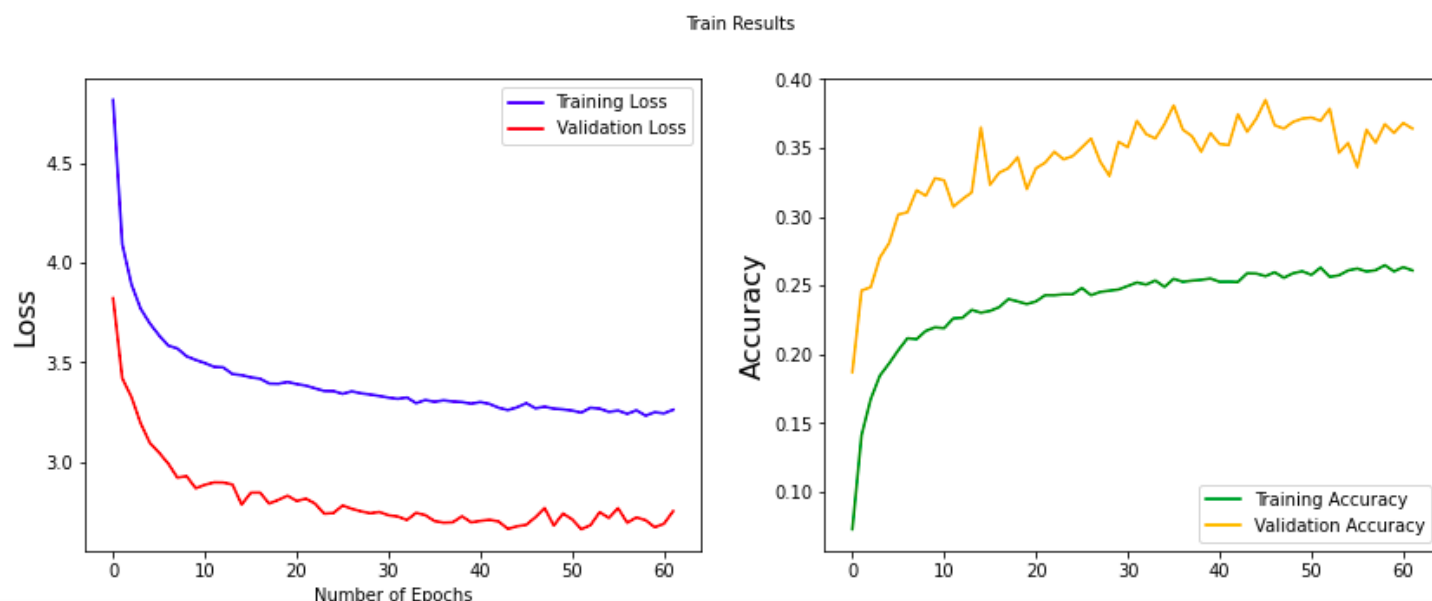


Figura 9: Variação da acurácia e da *loss* do modelo com transfer-learning InceptionV3

Train accuracy of the model: 0.2610251307487488

Train loss of the model: 3.261492967605591

Validation accuracy of the model: 0.36399999260902405

Validation loss of the model: 2.7527670860290527

Test Loss: 2.6706104278564453

Test Accuracy: 0.3440000116825104

Estas acurácias mais baixas podem dever-se à falta de recursos computacionais principalmente no modelo *VGG16* e uma ligeira falha na escolha de parâmetros. No modelo *Inception* poderá ter a ver com o congelamento de todas as camadas, uma vez que parece, pelos gráficos, que o modelo já convergiu e estagnou, apesar desse congelamento ocorrer em grande parte, por falta de recursos computacionais e tempo para realizar testes de apenas congelar as últimas 5 camadas ao contrário de todas elas.

5.3 InceptionResNetV2

InceptionResNetV2 é uma CNN que foi treinada com mais de um milhão de imagens do *ImageNet dataset*. O modelo tem 164 camadas e consegue classificar imagens em 1000 categorias de objeto, como o teclado, o rato, o lápis e muitos animais. [2]

Esta rede necessita de no mínimo como *input* imagens de 75x75 píxeis, então logo no início do *notebook* alterou-se o *target size* de (56,56) para (75,75).

O modelo final consistiu no modelo pré-treinado e depois adicionando o seguinte modelo:

```

model=Sequential()
model.add(base_model)
model.add(Dropout(0.5))
model.add(Flatten())
model.add(BatchNormalization())
model.add(Dense(2048,kernel_initializer='he_uniform'))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(1024,kernel_initializer='he_uniform'))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(250,activation='softmax'))

```

De todos os modelos com a técnica *transfer learning* que se experimentou, este foi o que obteve melhores resultados:

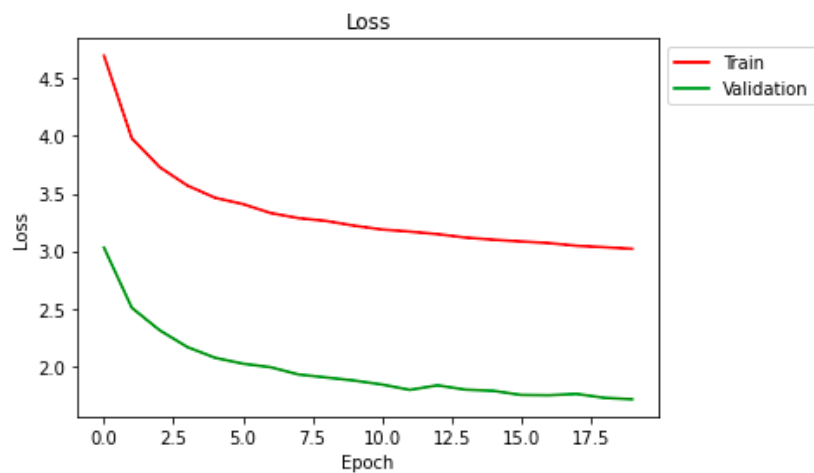


Figura 10: Variação da *loss* do modelo com transfer-learning InceptionResNetV2

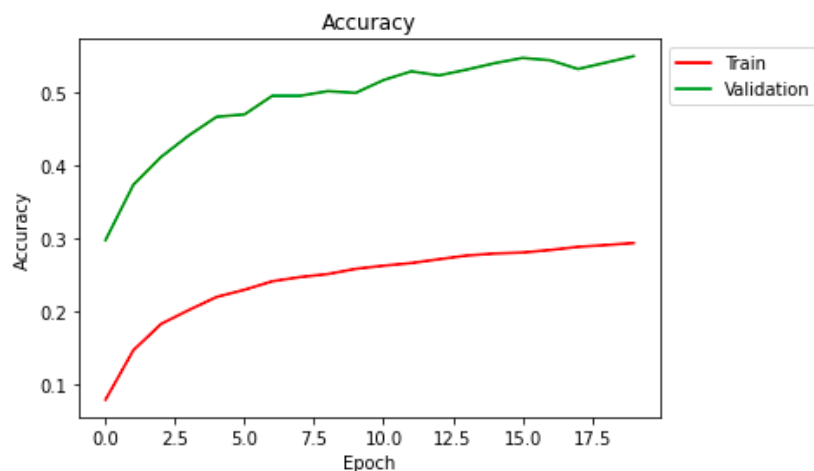


Figura 11: Variação da acurácia do modelo com transfer-learning InceptionResNetV2

Test Loss: 1.6864383220672607

Test Accuracy: 0.5616000294685364

5.4 Testes e previsões

Como o melhor modelo com *transfer learning* a nível de val loss (que foi sempre o que se esteve a tentar melhorar) foi o modelo *InceptionResNetV2* com uma *test loss* que ronda os 1.69, que é bastante menor, comparando com a *test loss* do melhor modelo anterior (modelo da versão 2) que é 2.23, considerou-se que o melhor modelo obtido até agora é o modelo com *transfer learning InceptionResNetV2*.

Com este modelo e com a função *predict*, fez-se algumas previsões sobre certas imagens de aves que se encontram no conjunto de testes e estes foram os resultados obtidos:

69.3 % chances are there that the bird is MASKED BOOBY

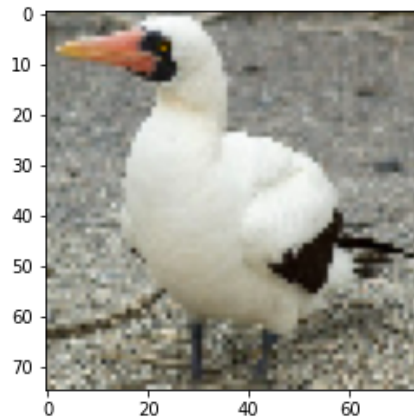


Figura 12: Predict MASKED BOOBY

85.99 % chances are there that the bird is PAINTED BUNTIG

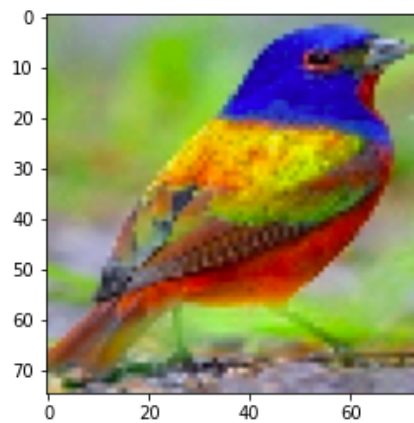


Figura 13: Predict PAINTED BUNTIN

Apesar da qualidade de imagem não ser a melhor (uma vez que no início se alterou a mesma para a obtenção de um maior poder computacional e recursos), ainda é possível para o algoritmo determinar qual é a probabilidade de realmente acertar na classe da imagem que lhe estamos a fornecer. Nestes dois exemplos, ambas as imagens que se forneceu foram classificadas corretamente, com uma certeza de 69.3 no primeiro e 85.99 no segundo.

6 Algoritmo genético

Para conseguir uma otimização de parâmetros, usou-se um algoritmo genético em função do melhor modelo obtido, ou seja o modelo *InceptionResnetV2*.

Os parâmetros (genes) a otimizar (de acordo com a pesquisa realizada e já referida acima) são:

- Função de ativação: elu, relu
- Função de otimização: adam, sgd, adagrad, rmsprop
- Função de loss: categorical_crossentropy
- Batch Size = 32, 64, 128, 256
- Número de nodos das camadas: 512, 1024, 2048
- Número de camadas: 2, 3, 4
- Número de filtros: 16, 32, 64, 128
- Tamanho do kernel: (3,3), (4,4)
- Tamanho do pooling: (2,2), (3,3)
- Dropout: 0.1, 0.2, 0.4, 0.5
- Learning Rate: 1, 0.1, 0.01, 0.001, 0.0001

Devido aos recursos disponíveis, foi estipulada uma população de 3 soluções, sendo que o algoritmo irá parar ao fim de 3 execuções também, existindo assim 9 indivíduos.

6.1 Funções de Fitness e de Seleção

Para cálculo de *fitness* foi criada uma função que dependia dos valores de *accuracy* do *dataset test*. Não se usou os valores da *loss* neste ponto do trabalho ao contrário de todos os outros modelos porque se entende que o modelo agora treinado já não estará em *overfitting* ou *underfitting* independentemente dos parâmetros utilizados, fazendo assim com que se procure apenas pela melhor acurácia do modelo através de *fine-tuning* dos parâmetros.

Para o processo de seleção dos indivíduos a criar descendência foi escolhido um processo baseado numa "Roda da Sorte": **Stochastic universal sampling**. [3] Cada indivíduo terá uma probabilidade de ficar na nova geração, mas esta probabilidade varia consoante o *fitness* de cada um. Este método foi escolhido porque permite a existência de possíveis soluções menos boas que contenham talvez alguns bons genes para passar a outro descendente e criar uma melhor solução.

6.2 Funções de Crossover e de Mutação

Após selecionada metade da geração a manter para gerar descendentes, o processo de *crossover* é então executado em cada par possível desta. Este *crossover* é executado usando **Uniform Crossover**, onde, para cada gene do descendente, é escolhido aleatoriamente de qual dos pais irá recebê-lo.

E em termos de mutação definiu-se que 70% de probabilidade de ocorrer era equilibrado, uma vez que, novamente devido à escassez de recursos, apenas se

tem 3 soluções. Esta mutação ocorre em apenas um dos genes que possui, tendo todas estas a mesma probabilidade de ocorrer. Assim pode-se ver que o processo de otimização da *pipeline* passou pelos pontos pretendidos na imagem disponível no enunciado:

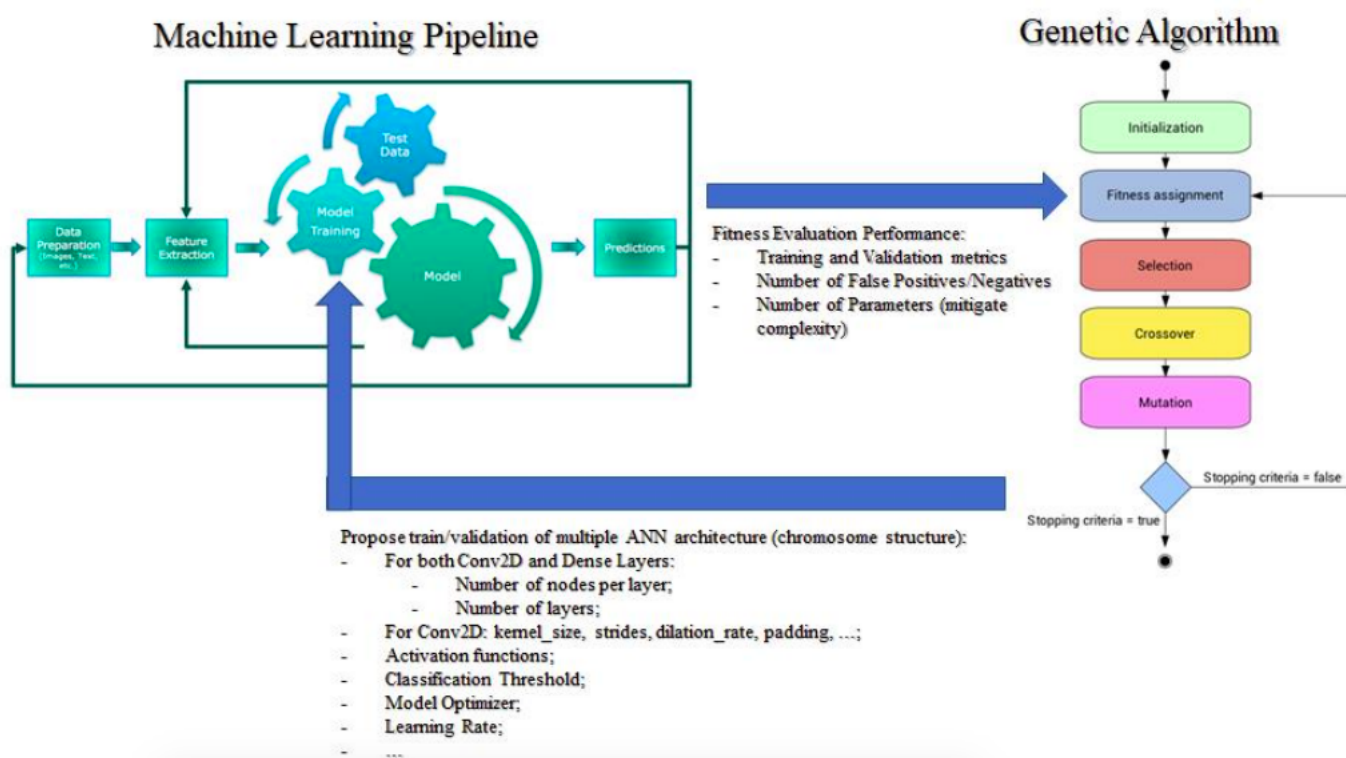


Figura 14: Pipeline do processo de otimização

6.3 Criação do modelo

Como anteriormente referido, o modelo criado teve como base o modelo *InceptionResnetV2*, com camadas adicionais. Inicialmente, pensou-se que adicionar camadas Convolucionais e de *MaxPolling* iria ser uma vantagem, mas depois de alguns modelos treinados assim com vários tamanhos do kernal e tamanhos de pooling verificou-se que o modelo *InceptionResnetV2* já teria suficiente camadas *Conv2* e *MaxPool* e que apenas as camadas mais regulares e normais do tipo *Dense* ajudariam à aprendizagem do modelo.

Assim, depois de se adicionar o modelo base ao modelo final adiciona-se um *Dropout*, *Flatten* e *BatchNormalization()* com intuito de que a aprendizagem obtida pelo modelo de base não fique com demasiado ruído (aprendeu demasiado ou de forma errada com o dataset de treino e não consegue transpor para outros datasets). E a seguir a esta fase inicial, coloca-se um ciclo para determinar quantas camadas o modelo deve ter (visto pelo gene que indica o número de camadas, começando por ter 2 camadas) seguido de *BatchNormalization* e *Dropout* com o intuito de fazer o modelo aprender de forma concisa.

A seguir encontra-se o modelo treinado:

```
def create_model(options : Individual):
model=Sequential()
#base_model = Inception_resnet_v2
model.add(base_model)
#model.add(Dropout(dropout))
#model.add(Conv2D(options.n_filters, kernel_size=options.kernel_sizes,
#activation=options.activation_function, input_shape=(56,56,3),padding='Same'))
#model.add(MaxPool2D(pool_size=options.pool_sizes, padding='Same'))
model.add(Dropout(options.dropouts))
#Last layer
model.add(Flatten())
model.add(BatchNormalization())
for i in range(options.n_layers):
    model.add(Dense(options.n_nodes,activation=options.activation_function))
    model.add(BatchNormalization())
    model.add(Dropout(options.dropouts))

model.add(Dense(250, activation='softmax'))

if options.optimizer == "adam":
    optimizer = optimizers.Adam(learning_rate = options.lr)
elif options.optimizer == "rmsprop":
    optimizer = optimizers.RMSprop(learning_rate = options.lr)
elif options.optimizer == "sgd":
    optimizer = optimizers.SGD(learning_rate = options.lr, momentum = 0.9)
elif options.optimizer == 'adagrad':
    optimizer = optimizers.Adagrad(learning_rate = options.lr)

# compile the keras model
model.compile(loss=options.loss, optimizer=optimizer, metrics=['accuracy'])
```

6.4 Resultados

Correndo então o algoritmo genético, verifica-se que ao fim de 9 iterações conseguiu-se em algumas iterações seguidas indivíduos com resultados satisfatórios, acabando por piorar mais tarde mas parecendo que se se aumentasse o número de iterações os resultados seriam muito melhores (algo que não foi feito, uma vez que o algoritmo, já usando apenas 30 epochs e com uma paciência de 6 a nível de minimizar a *val_loss*, demorou 10 horas e 18 minutos para correr no *GoogleCollab* usando GPU, e o limite máximo do mesmo é 12 horas. Apesar do algoritmo não ter convergido numa única solução (o que faz sentido dado que muitas arquiteturas vão dar valores semelhantes) a seguir apresenta-se as métricas usadas por cada indivíduo e os resultados obtidos que também se encontram no ficheiro *individuals.txt* já demonstra uma aprendizagem eficiente do algoritmo:

```
-----
Generation 0 ID 0
Fitness Score: 0.4519999921321869
```


Filters: 32
Kernel_size: (3, 3)
Pool_size: (2, 2)
Layers: 3
Nodes: 1024
Batch Size: 256
Dropout: 0.4
Learning-Rate: 0.001
Activation_Function: elu
Optimizer: sgd
Losse Function: categorical_crossentropy
ACC: 0.4519999921321869

Generation 0 ID 1
Fitness Score: 0.4431999921798706
Filters: 32
Kernel_size: (4, 4)
Pool_size: (2, 2)
Layers: 2
Nodes: 1024
Batch Size: 64
Dropout: 0.5
Learning-Rate: 0.01
Activation_Function: relu
Optimizer: adagrad
Losse Function: categorical_crossentropy
ACC: 0.4431999921798706

Generation 0 ID 2
Fitness Score: 0.527999997138977
Filters: 32
Kernel_size: (3, 3)
Pool_size: (3, 3)
Layers: 3
Nodes: 512
Batch Size: 256
Dropout: 0.2
Learning-Rate: 0.01
Activation_Function: elu
Optimizer: sgd
Losse Function: categorical_crossentropy
ACC: 0.527999997138977

Generation 1 ID 3
Fitness Score: 0.47999998927116394
Filters: 32
Kernel_size: (3, 3)

Pool_size: (2, 2)
Layers: 3
Nodes: 1024
Batch Size: 256
Dropout: 0.4
Learning-Rate: 0.01
Activation_Function: elu
Optimizer: sgd
Losse Function: categorical_crossentropy
ACC: 0.47999998927116394

Generation 1 ID 4
Fitness Score: 0.520799994468689
Filters: 32
Kernel_size: (3, 3)
Pool_size: (2, 2)
Layers: 2
Nodes: 512
Batch Size: 256
Dropout: 0.2
Learning-Rate: 0.01
Activation_Function: elu
Optimizer: sgd
Losse Function: categorical_crossentropy
ACC: 0.520799994468689

Generation 1 ID 5
Fitness Score: 0.3968000113964081
Filters: 32
Kernel_size: (4, 4)
Pool_size: (2, 2)
Layers: 3
Nodes: 1024
Batch Size: 256
Dropout: 0.5
Learning-Rate: 0.001
Activation_Function: elu
Optimizer: sgd
Losse Function: categorical_crossentropy
ACC: 0.3968000113964081

Generation 2 ID 6
Fitness Score: 0.520799994468689
Filters: 32
Kernel_size: (3, 3)
Pool_size: (2, 2)
Layers: 2

```
Nodes: 512
Batch Size: 256
Dropout: 0.2
Learning-Rate: 0.01
Activation_Function: elu
Optimizer: sgd
Losse Function: categorical_crossentropy
ACC: 0.520799994468689
-----
```

```
-----
Generation 2 ID 7
Fitness Score: 0.5016000270843506
Filters: 32
Kernel_size: (4, 4)
Pool_size: (2, 2)
Layers: 3
Nodes: 512
Batch Size: 256
Dropout: 0.2
Learning-Rate: 0.001
Activation_Function: elu
Optimizer: sgd
Losse Function: categorical_crossentropy
ACC: 0.5016000270843506
-----
```

```
-----
Generation 2 ID 8
Fitness Score: 0.3887999951839447
Filters: 32
Kernel_size: (3, 3)
Pool_size: (2, 2)
Layers: 3
Nodes: 1024
Batch Size: 256
Dropout: 0.5
Learning-Rate: 0.001
Activation_Function: elu
Optimizer: sgd
Losse Function: categorical_crossentropy
ACC: 0.3887999951839447
-----
```

Assim, podemos concluir que o melhor modelo obtido ocorreu na geração 0 com o indivíduo 2 com uma acurácia de 0.5279999997138977, a utilizar 32 filtros, com o tamanho do kernel (3,3) e tamanho de pooling (3,3), com 3 camadas e 512 nodos, com um *batch_size* de 256, um *Dropout* de 0.2, um *Learning Rate* de 0.01 e sendo a sua função de ativação a *elu* e o seu otimizador o *sgd*.

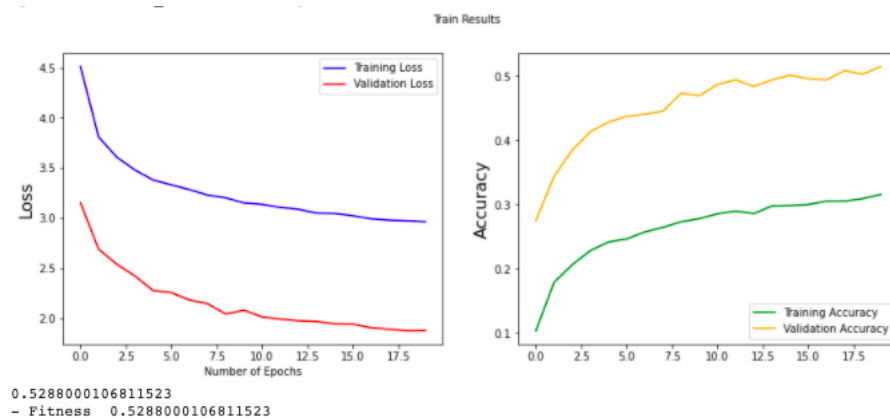


Figura 15: Melhor resultado

7 Conclusão

Apesar do algoritmo genético não ter apresentado resultados superiores (apenas parecidos) comparado com quando não é usado isto deve-se a uma escassez de tempo e de recursos para conseguir testa-lo com um grande número de *epochs* (om o mesmo número de *epochs usado para cada modelo CNN*) para o mesmo conseguir convergir com uma paciência maior também. Assim, os resultados obtidos com *transfer-learning* vão de encontro com o estado de arte que se fez na primeira parte do trablho, apenas ligeiramente mais baixos, devido ao *dataset* se encontrar com 80% de imagens de pássaros masculinas e apenas 20% femininas (diferente dos *dataset* encontrados na pesquisa feita), fazendo com que o modelo possa não aprender tão bem em imagens de espécies femininas. Porém ao usar o *predict* do modelo, ou seja a prever a classificação de uma espécie para o *dataset* de testes obtem-se grandes resultados com 69.3% de chance de prever corretamente e a chegar mesmo a 85.99% de chance, o que no caso deste trabalho são valores bastante elevados e satisfatórios.

Conclui-se que neste projeto todos os conhecimentos dados nas aulas foram bem aplicados e bastante consolidados, tendo que se entender de forma bastante profunda todos os parâmetros que são usados nos modelos CNN e no Algoritmo Genético e verificando o quão complicado é treinar modelos com *Deep Learning* de grande escala e com uma enorme robustez. Assim, considera-se que o trabalho foi bem desenvolvido e que os conhecimentos aprendidos com o mesmo são uma mais valia para o futuro.

Referências

- [1] <https://www.analyticsvidhya.com/blog/2020/02/learn-image-classification-cnn-convolutional-neural-networks-3-datasets>
- [2] <https://www.kaggle.com/gauravrajpal/bird-species-classification-v1-2-incepresnet-95>

- [3] Krishnaveni, A. (2019). A Survey on Natural Inspired Computing (NIC): Algorithms and Challenges. Global journal of computer science and technology.
- [4] Castro, L. (2007). Fundamentals of natural computing: an overview. Physics of Life Reviews, 4, 1-36.
- [5] Engelbrecht A. (2007), Computational Intelligence: An Introduction, Wiley Sons. ISBN 0-470-84870-7.