Trabalho Prático
Computação Natural

Hugo Faria

Universidade do Minho, Departamento de Informática

Resumo O presente relatório tem como objectivo a preparação e desenvolvimento de um modelo preditivo usando *Python* que classifique espécies de aves que aparecem numa imagem através da implementação de algoritmos CNN.

Keywords: Transfer learning, Algoritmo genético, CNN

Conteúdo

1 Introdução

Este relatório é referente ao primeiro trabalho prático da unidade curricular de Computação Natural do perfil de Sistemas Inteligentes. O trabalho consiste na preparação e análise dum *dataset* de espécies de aves, seguido de treino e validação de redes neuronais, e por fim na aplicação de um algoritmo genético para optimizar o modelo.

Numa primeira fase será então feita uma explicação sobre o pré processamento das imagens. Em seguida, passaremos à construção de várias redes neuronais na procura da melhor solução.

Depois deste processo, iremos falar sobre a utilização de um algoritmo genético usado de maneira a conseguir obter um melhor resultado com as redes neuronais.

2 Primeira versão - CNN simples

Uma CNN, ou rede neuronal convolucional, é um algoritmo de *deep learning* que consegue receber imagens, associar importância a vários elementos da imagem e diferenciar entre os mesmos. O pré-processamento desta é bastante mais baixo comparado com outros algoritmos de classificação.

2.1 Pré-processamento

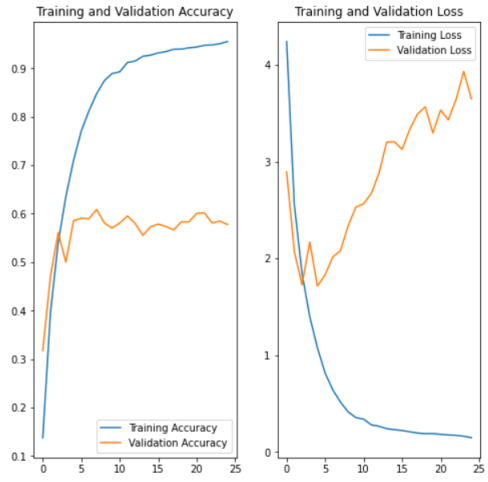
O nosso *dataset* de imagens consiste em imagens de 250 espécies diferentes de aves de tamanho 224 x 224 com 3 *color channels* no formato jpg. Estas imagens estão divididas em *train*(35215 imagens),*valid*(1250 imagens) e *test*(1250 imagens).

A primeira coisa a ser feita é normalizar os valores de *input*, uma vez que os valores atuais das imagens estão no intervalo de 0 a 255 sendo que o ideal para uma rede neuronal é de 0 a 1. De seguida, criam-se os *dataset* de treino, validação e teste usando a função *flow_from_directory*. Cada *dataset* é gerado com um *batch_size* de 32, mantendo o tamanho de 224 x 224 das imagens e fazendo um *shuffle*.

2.2 Modelo

Foram feitos vários modelos para testar resultados, seja com diferentes camadas convolucionais, com diferentes camadas densas, utilização de *data augmentation* ou de *dropout*. Todos os modelos foram corridos com 25 epochs.

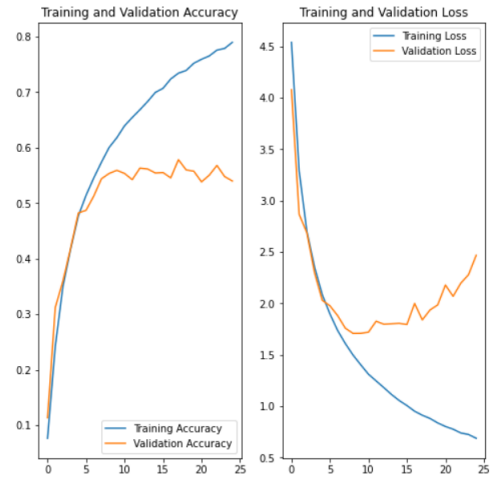
Modelo com 3 *layers* convolucionais e 3 densas O primeiro modelo a ser feito foi um simples modelo com três *layers* convolucionais e três densas, duas com tamanho 100 e uma com tamanho 250, percorrido com *strides* de 2 por 2 e com *padding 'same'*. Obteve-se uma perda de 3.25 e uma acurácia de 0.60.




```
Test Loss: 3.249337911605835
Test accuracy 0.6015999913215637
```

Os resultados deste modelo não foram os melhores, portanto serão feitas várias mudanças.

Modelo com 4 *layers* convolucionais e 3 densas O segundo modelo a ser feito consistiu em adicionar uma *layer* convolucional.

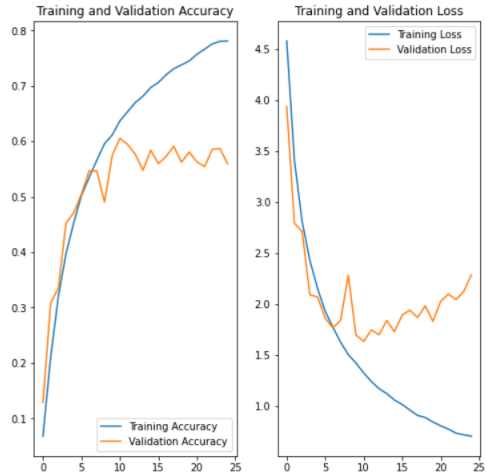



```
Test Loss: 2.2281594276428223
Test accuracy 0.5776000022888184
```

Verificamos uma ligeira descida na acurácia do teste. No entanto, esta pode ser causada pela maneira que a função *flow_from_directory* usada para criar um

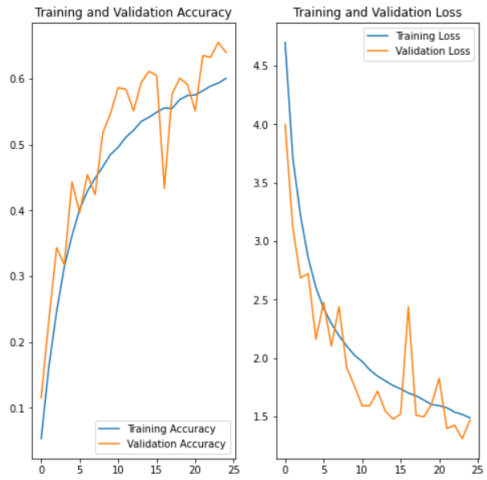
dataset com as imagens funciona. Como há uma descida na perda de 1 valor, concluiu-se que houve uma ligeira melhoria no modelo. Veremos então o que acontece se adicionar mais uma *layer*.

Modelo com 5 *layers* convolucionais e 3 densas Uma vez que se verificou uma melhoria nos resultados, foi adicionada mais uma *layer* convolucional.



Voltou-se a verificar uma ligeira melhoria na perda, tendo a acurácia voltado aos valores do primeiro modelo. Concluimos assim que adicionar *layers* convolucionais foi benéfico ao modelo e passaram-se a testar outras opções sobre o mesmo.

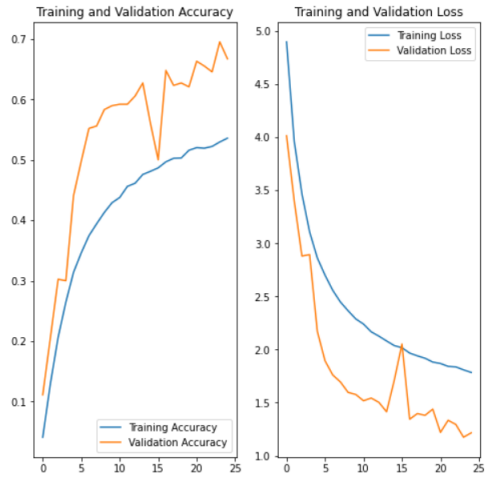
Modelo com 5 *layers* convolucionais e 3 densas com *Data Augmentation* Como se pôde ver no gráfico anterior, a linha da validação está bastante abaixo da linha de treino. De maneira a evitar isto, foi adicionada *data augmentation* que consiste em criar nova informação baseada nas imagens do *dataset*, criando neste caso imagens viradas horizontalmente, imagens com zoom e imagens que sofreram uma rotação.




```
Test Loss: 1.340004801750183
Test accuracy 0.6312000155448914
```

Obtivemos assim melhores resultados do que em qualquer um dos modelos anteriores, apresentando uma descida ainda acentuada na perda e uma pequena subida na acurácia.

Modelo com 5 *layers* convolucionais e 3 densas com dropout De maneira a evitar um possível *overfitting* causado pela *data augmentation*, aplicou-se *dropout*, que consiste em não usar alguns do neurónios usados de forma aleatória.




```
Test Loss: 1.1264618635177612
Test accuracy 0.6815999746322632
```

Voltamos a verificar uma melhoria nos resultados, tendo a perda descido ligeiramente e a acurácia aproximado de 70

Através dos resultados obtidos com estes modelos podemos ver que esta ainda não é a melhor maneira de obtermos uma rede neuronal apropriada para o *dataset* dado. Como consequência das conclusões tiradas, passou-se a adoptar outra técnica, a de *transfer learning*.

3 Segunda versão - Modelo com *transfer learning*

Transfer learning consiste na reutilização de modelos pré treinados num novo problema. É uma técnica muito usada em *deep learning* para poder treinar grandes redes neuronais com comparativamente menos informação. De modo a conseguir obter melhores resultados que os verificados anteriormente, aplicou-se a mesma.

3.1 Pré-processamento

No pré-processamento das imagens utilizou-se na mesma a normalização dos valores, seguido desta vez da função de pré processamento do modelo de *transfer learning* usado, ou seja, VGG16, VGG19 ou Resnet50.

3.2 Modelos

A cada um dos modelos foram adicionadas 3 camadas densas de tamanho 250. Em alguns dos modelos foi também adicionado um *callback* que consiste em reduzir o *learning rate* quando a perda da validação estagna durante quatro epochs de maneira a que o modelo possa ter melhores resultados.

Uma vez que estes modelos levam mais tempo a ser treinados, escolheu-se 15 epochs como o número a ser usado uma vez que, em comparação com modelos de 10 a 30 epochs, foi considerando suficiente para avaliar os modelos em comparação com as versões anteriores.

VGG16 normal Usando uma VGG16 normal com as *layers* de treino congeladas conseguimos obter resultados de 0.56 na perda e de 0.856 na acurácia

```
Test Loss: 0.5684896111488342
Test accuracy 0.8560000061988831
```

Comparando aos resultados sem *transfer learning* obtivemos uma melhoria substancial.

VGG16 com callback Aplicando agora o *callback* ao VGG16 obtivemos resultados de 0.57 na perda e de 0.877 na acurácia. Verifica-se assim uma ligeira melhoria na acurácia mas nada de substancial.

```
Test Loss: 0.5703545212745667
Test accuracy 0.8776000142097473
```

VGG19 normal Usando agora uma VGG19 normal com as *layers* de treino congeladas conseguimos obter resultados de 0.50 na perda e de 0.861 na acurácia Os resultados são similares à utilização do VGG16 normal, não notando assim nenhuma vantagem em utilizar antes este modelo.

```
Test Loss: 0.5033891797065735
Test accuracy 0.8615999817848206
```

VGG19 com callback Voltando agora a aplicar o *callback*, desta vez no modelo VGG19, os resultados já não foram tão bons, obtendo 0.728 na perda e 0.796 na acurácia.

```
Test Loss: 0.7289878726005554
Test accuracy 0.7968000173568726
```

Deveria-se testar mais vezes este modelo para se poder dizer com certeza se este é realmente pior ou não, no entanto, como esta versão ainda não é a versão com um algoritmo genético não foi considerado relevante estar a garantir a viabilidade do método para já.

Tentou-se ainda utilizar o modelo Resnet50, mas os resultados estavam a ser bastante baixos (acurácias de 0.004) por isso abandonou-se a ideia.

4 Terceira versão - Uso de algoritmos genéticos

4.1 Algoritmo Genético

Os algoritmos genéticos são algoritmos aleatórios evolucionários baseados na teoria da evolução de Darwin. Estes consistem em trabalhar sobre populações de soluções do dado problema. Cada solução é chamada de indivíduo e cada indivíduo é composto por diferentes cromossomas que correspondem aos parâmetros usados na definição do mesmo. Cada indivíduo apresenta também o chamado

valor de *fitness* que representa a qualidade do indivíduo e que vai ser o critério usado na escolha dos cromossomas que iram passar para a próxima geração.

De uma forma prática, o algoritmo consiste em, primeiro, formar diferentes indivíduos com diferentes cromossomas que constituem uma geração. De seguida, escolher os melhores indivíduos, ou seja, os com melhor *fitness*, e formar uma *mating pool*. Esta *mating pool* passa a ser constituída por indivíduos pais dos quais se irá escolher dois para formar um novo indivíduo com uma mistura dos seus cromossomas. Durante esta formação é possível a ocorrência duma mutação, ou seja, um dos cromossomas pode ser alterado aleatoriamente, Assim que se formarem vários novos indivíduos, substitui-se a geração anterior pela nova.

4.2 Estrutura

No algoritmo genético criado, cada individuo é composto pelos seguintes cromossomas:

- **Tipo :** Corresponde ao tipo de modelo de *transfer learning* usado no modelo, ou seja, VGG16, VGG19 ou Resnet50.
- **Dense_num :** Corresponde ao número de *dense layers* usadas, varia entre 1 a 4.
- **Dense_val :** Corresponde ao tamanho das *dense layers* usadas, pode ser ou 250 ou 512.
- **Batch_size :** Corresponde ao tamanho da *batch* ao recolher as imagens.
- **Learn_rate :** Corresponde ao tamanho da *learn rate* do modelo, pode variar entre 0.1, 0.01, 0.001, 0.0001 e 0.00001.
- **Learn_rate_reduce :** Indica se foi aplicado um *callback* que reduz o *learn rate* com um factor de 0.1 caso o valor da perda estagne durante quatro epochs.

Estes cromossomas estão representados numa classe chamada *Settings*

```
class Settings:

  def __init__(self,type,dense_num,dense_val,batch_size,learn_rate,learn_rate_reduce):
    self.type = type
    self.dense_num = dense_num
    self.dense_val = dense_val
    self.batch_size = batch_size
    self.learn_rate = learn_rate
    self.learn_rate_reduce = learn_rate_reduce
```

A classe settings está depois contida dentro doutra classe chamada CNN que contém também a *fitness* do modelo. Neste caso, a *fitness* corresponde à acurácia.

```
class CNN:
  def __init__(self,Settings,fitness):
    self.Settings = Settings
    self.fitness = fitness
```

Durante a primeira geração, são gerados diferentes conjuntos de cromossomas para cada elemento da geração, sendo depois chamada a função corresponde à iniciação da rede neuronal correspondente.

Assim que a primeira geração acaba, os vários modelos que foram guardados são mandados à função *mating*. Esta função recebe a lista de modelos e seleciona os dois com melhor fitness, devolvendo uma lista das *settings* dos mesmos

```
#Função que seleciona os melhores pais
def mating(list):
  best_id = 0
  second_best_id = 0
  for i in range(0,elements):
    if list[i].getFitness() > list[second_best_id].getFitness():
     if list[i].getFitness() > list[best_id].getFitness():
       best_id = i
     else:
       second_best_id = i
  parents = [list[best_id].getSettings(),list[second_best_id].getSettings()]
  return parents
```

De seguida, os dois pais são mandados à função *crossover* que vai gerar quatro conjuntos de cromossomas que podem ser herdados pelos filhos

```
#Função que gera as possiveis settings dos filhos
def crossover(cnnList):
  settings_list = []
  settings_list.append(Settings(cnnList[0].type,cnnList[1].dense_num,
  cnnList[0].dense_val,cnnList[1].batch_size,cnnList[0].learn_rate,
  cnnList[1].learn_rate_reduce))
  settings_list.append(Settings(cnnList[1].type,cnnList[0].dense_num,
  cnnList[1].dense_val,cnnList[0].batch_size,cnnList[1].learn_rate,
  cnnList[0].learn_rate_reduce))
  settings_list.append(Settings(cnnList[0].type,cnnList[0].dense_num,
  cnnList[0].dense_val,cnnList[1].batch_size,cnnList[1].learn_rate,
  cnnList[1].learn_rate_reduce))
  settings_list.append(Settings(cnnList[1].type,cnnList[1].dense_num,
  cnnList[1].dense_val,cnnList[0].batch_size,cnnList[0].learn_rate,
  cnnList[0].learn_rate_reduce))
  return settings_list
```

Após isto, os conjuntos são mandados à função *mutation* que vai, ou não, mudar algum dos cromossomas. A chance de tal acontecer é dada pela variável *mutation_rate*

```
def mutation(settings_list):
  for i in range(0,4):
    probability = random.uniform(0.0,1.0)
    if probability < mutation_rate:
```

```
    rand_setting = random.randint(0,5)
    if rand_setting == 0:
      settings_list[i].setType(random.choice(['VGG16','VGG19','Resnet']))
    if rand_setting == 1:
      settings_list[i].setDense_num(random.randint(1,4))
    if rand_setting == 2:
      settings_list[i].setDense_val(random.choice([250,512]))
    if rand_setting == 3:
      settings_list[i].setBatch_size(random.choice([32,64]))
    if rand_setting == 4:
      settings_list[i].setLearn_rate(random.choice([0.1,0.01,0.001,0.0001,0.00001]))
    if rand_setting == 5:
      settings_list[i].setLearn_rate_reduce(random.choice(['yes','no']))
  return settings_list
```

Depois disto geram-se os vários filhos escolhendo aleatoriamente um conjunto de cromossomas dos gerados, repetindo-se o ciclo até chegar ao número de gerações pretendidas.

```
def init(elements,generations):
  models = []
  mating_cnns = []
  crossover_settings = []
  g = 0
  for g in range(g,generations):
   print("Começou a geração: ",g+1)
   epochs = 5
   if g != 0 :
    mating_cnns = mating(models)
    crossover_settings = crossover(mating_cnns)
    crossover_settings_mutated = mutation(crossover_settings)
   for n in range(0,elements):
     print("CNN número: ",n+1)
     if g == 0:
      type = random.choice(['VGG16','VGG19','Resnet'])
      dense_num = random.randint(1,4)
      dense_val=random.choice([250,512])
      batch_size=random.choice([32,64])
      learn_rate=random.choice([0.1,0.01,0.001,0.0001,0.00001])
      learn_rate_reduce=random.choice(['yes','no'])
      settings = Settings(type,dense_num,dense_val,batch_size,learn_rate,learn_rate_reduce)
     else:
      rand_setting = random.randint(0,3)
      settings = crossover_settings_mutated[rand_setting]
     if settings.getType() == 'VGG16':
       print("Starting VGG16")
       models.append(startVGG16(epochs,settings))
```

```
    if settings.getType() == 'VGG19':
      print("Starting VGG19")
      models.append(startVGG19(epochs,settings))
    if settings.getType() == 'Resnet':
      print("Starting Resnet")
      models.append(startResnet(epochs,settings))
    print("------------------------------")
  print("------------------------------------------------------------")
```

4.3 Análise de resultados

Devido às limitações de tempo e hardware, foi corrido o algoritmo genético com apenas cinco elementos, com cinco epochs e três gerações. No entanto, para obter os resultados mais precisos com o algoritmo, deveria-se correr o mesmo com um maior número de elementos, gerações e epochs, escolhendo mais pais de cada geração, obtendo assim mais combinações e podendo assim chegar a um resultado final mais certo. Poderia-se também adicionar novos tipos de cromossomas, como por exemplo o tipo do optimizador e o factor pelo qual se vai diminuir o *learn rate* quando aplicado o *callback*.

No final desta execução mais básica, o melhor modelo obtido apresentava uma *fitness* de 0.83 ao fim de cinco epochs. Este modelo apresentava a seguinte configuração:

- **Tipo :** VGG16.
- **Dense_num :** 2.
- **Dense_val :** 512.
- **Batch_size :** 64.
- **Learn_rate :** 0.001.
- **Learn_rate_reduce :** Sim.

De modo a melhorar ainda este modelo, foi aplicado *fine tuning* ao mesmo. Após descongelar todas as camadas de treino do modelo, foi corrido um *fit* do mesmo com cinco epochs e com uma *learn rate* de 0.0001, obtendo uma perda de 0.31 e uma acurácia de 0.928, o que se pode considerar bons resultados para um algoritmo que correu com pouca variação.

```
Test Loss: 0.318600594997406
Test accuracy 0.9287999868392944
```

5 Conclusão

A realização deste trabalho permitiu um aprofundamento do conhecimento na utilização de redes neuronais, desde a análise de *datasets* até à utilização de algoritmos mais avançados, como os genéticos.

Como extensão do trabalho, deveria-se então aumentar a complexidade do algoritmo genético criado, podendo assim, como já foi dito, obter resultados mais precisos e eventualmente mais próximos duma acurácia de 100