Model S without data augmentation

Arquitetura

Com base no nosso conhecimento adquirido ao longo das aulas e com alguma pesquisa, chegámos à conclusão que a melhor arquitetura a utilizar é a seguinte:

1º Camada convolucional

- 32 filtros com dimensão de 3x3;
- Função de ativação ReLU, pois foi aquela com que mais nos familiarizamos nas aulas e com base na nossa pesquisa nos pareceu a mais indicada;
- MaxPooling com dimensão 2x2 para conseguirmos reduzir a dimensão e extrair as características mais importantes

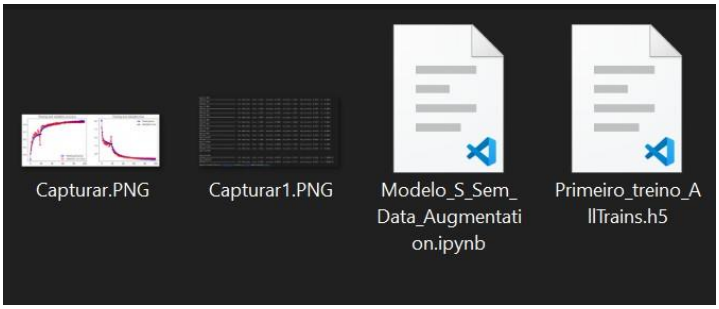
2º Camada convolucional

- As características são praticamente similares à primeira camada, mas decidimos aumentar o número de filtros para 64 que com o aumento da profundidade da rede conseguíssemos extrair mais características

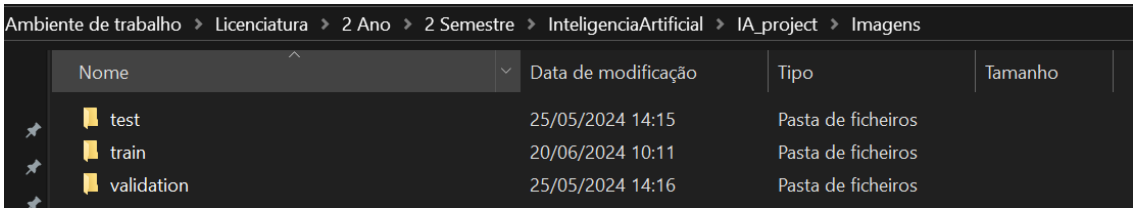
Após estas duas camadas, replicámo-las, mas com o dobro dos filtros da segunda camada convolucional, acreditamos ter sido um exagero, mas como a nossa inexperiência falou mais alto, tivemos de começar por algum lado.

```
inputs = keras.Input(shape=(IMG_SIZE, IMG_SIZE, 3))

#Layers

x = layers.Conv2D(filters=32, kernel_size=3, activation="relu")(inputs)
x = layers.MaxPooling2D(pool_size=2)(x)

x = layers.Conv2D(filters=64, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)

x = layers.Conv2D(filters=128, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)

x = layers.Conv2D(filters=128, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)

x = layers.Flatten()(x)
x = layers.Dense(512, activation="relu")(x)
outputs = layers.Dense(10, activation="softmax")(x)

model = keras.Model(inputs=inputs, outputs=outputs)
```

Fizemos cerca de 10 treinos com 70 a 100 épocas cada, onde em cada treino guardamos o modelo treinado (h5), o código e uma print do gráfico.



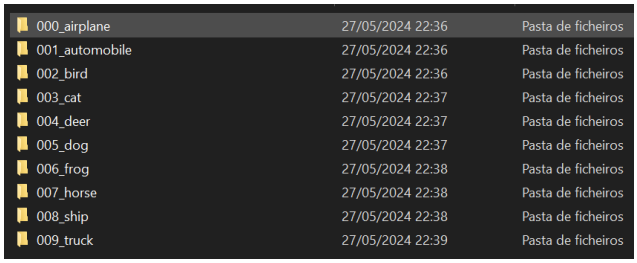
Datasets

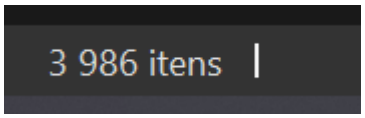


Nos processos de treino testámos utilizar um *dataset*, com todas as imagens de cada *label* todas na mesma pasta e testámos treinar o modelo com cada pasta individualmente, ou seja, treino 1, treino 2, treino 3, treino 4.

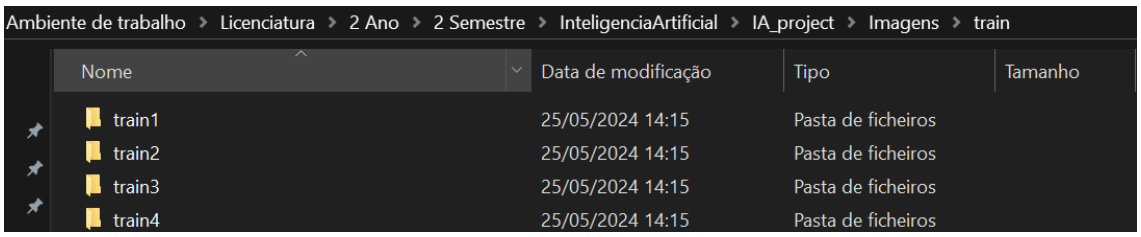
Todas as imagens na mesma pasta

Ou seja, cada pasta com aproximadamente 4000 imagens





Iterar pelas diferentes pastas




```python
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout, BatchNormalization
from tensorflow.keras.regularizers import l2
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import os
from tensorflow import keras

# Diretórios de dados
base_dir = 'Imagens/'
train_dirs = [os.path.join(base_dir, f'train/train{i}/') for i in range(1, 5)]
validation_dir = os.path.join(base_dir, 'validation')
test_dir = os.path.join(base_dir, 'test')
num_classes = 10
```

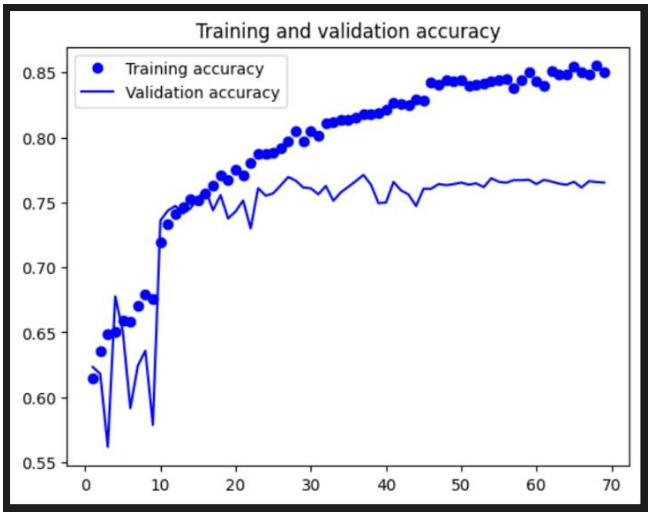
Ao longo dos treinos verificamos que a rede desempenhou melhor com a segunda forma de percorrer as imagens, ou seja, iterar pelas diferentes pastas, em vez de ter uma pasta com as 10 classes e cada classe todas as imagens.

O que retivemos ao longo dos treinos

Ao longo dos treinos testamos vários *hiperparâmetros*, começamos com um *batchsize* pequeno de 32, o que ao longo dos treinos se revelou algo importante pois, quanto maior era o *batchsize* mais estagnada a rede ficava. Pelo que percebemos, embora um *batchsize* maior gere um gradiente mais estável, por vezes pode não se benéfico, pois um certo nível de ruído pode ajudar o gradiente a escapar de mínimos locais.

Por volta do 5º treino percebemos que a rede estava com dificuldades em melhorar, o que nos fez voltar a mudar a arquitetura, desta vez aumentando o número de filtros da última camada convolucional de 128 para 512.

Com esta mudança começaram-se a ver melhores resultados, mas reparou-se que a rede começou com algum *overfitting*.



Após mais 2 treinos, decidimos voltar atrás na arquitetura e desta vez fomos para um valor entre os dois já testados, ou seja, nem 128, nem 512. A escolha para a nossa última camada foi de 256 filtros. Nos treinos seguintes notamos uma melhoria significativa na rede onde começamos a atingir valores de 79 de val_acc.

Dropout e BatchNormalization

Quando começamos a notar que a rede estava sempre nos 79 de val_acc e com algum *overfitting*, decidimos dar um *step up* e ir pesquisar, perceber e implementar formas de melhorar a rede e de reduzir o *overfitting*, onde descobrimos também uma técnica chamada de *callback*.

As técnicas que utilizamos, foram técnicas de *Dropout*, e o *BatchNormalization*.

Começamos com o Dropout, pois foi uma técnica utilizada para reduzir o *overfitting* que nos tinha ficado na cabeça durante as aulas práticas, em que no nosso entender consiste numa forma de regularização onde os neurónios são desativados aleatoriamente durante o processo de treino, para "forçar" a rede a aprender as características e a generalizá-las, em vez de as "decorar". Depois de implementarmos o Dropout, inserimos *BatchNormalization* na nossa rede, que é utilizado para acelerar o treino e melhorar a escalabilidade do modelo. Após concluirmos a pesquisa percebemos que a forma de funcionamento é relativamente simples, o *BatchNormalization* recolhe um "mini-batch", calcula a média e o desvio padrão para uma determinada camada, inicia o processo de normalização que é apenas subtrair a média e dividir pelo desvio padrão, o que vai originar numa distribuição com média 0 e desvio padrão 1. Ainda existe a aplicação de uma transformação linear, mas após percebermos como funciona, tudo ficou mais simples. Na nossa visão o que o *BatchNormalization* faz é, como a normalização dos mini-batch's é aleatória, essa adição de ruído não é algo previsível, ou seja, irá ajudar a regularizar o modelo.

Fizemos uso da técnica de regularização l2, esta técnica consiste na aplicação de uma penalidade aos pesos do modelo, esta penalização é proporcional à soma dos quadrados dos valores dos pesos. A penalização incentiva o modelo a manter pesos menores, o que resulta num modelo simples e menos propensos a *overfitting*.

```
model = Sequential()
model.add(Conv2D(32, (3, 3), padding='same', activation='relu', input_shape=(32, 32, 3)))
model.add(BatchNormalization())
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.4))

model.add(Conv2D(64, (3, 3), padding='same', activation='relu', kernel_regularizer=l2(0.01)))
model.add(BatchNormalization())
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.5))

model.add(Conv2D(128, (3, 3), padding='same', activation='relu', kernel_regularizer=l2(0.01)))
model.add(BatchNormalization())
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.5))

model.add(Flatten())
model.add(Dense(512, activation='relu', kernel_regularizer=l2(0.001)))
model.add(BatchNormalization())
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
```

Com a aplicação destas duas técnicas o nosso modelo começou a melhorar bastante e continuamos com mais treinos. Quando atingimos os 84% de val_acc, pareceu-nos um resultado bastante bom tendo em conta o nosso poder computacional e o número de treinos realizados, mas estávamos a sofrer com alguma *overfitting*. Para resolver este *overfitting*, decidimos apenas acrescentar uma última técnica de regularização que consiste na penalização da função *loss*, baseada na magnitude dos pesos. No nosso ponto de vista a utilização desta técnica faz todo o sentido pois estamos a incentivar a rede a manter os pesos pequenos para evitar assim o *overfitting*.

CallBack

A técnica de *callback* mencionada anteriormente, consiste em dois conceitos, o *early_stopping* e o *reduce_learingRate*.

O *early_stopping*, como o nome indica, consiste em especificarmos quantas épocas vamos esperar (patience) com o val_loss estagnado, até pararmos a execução.

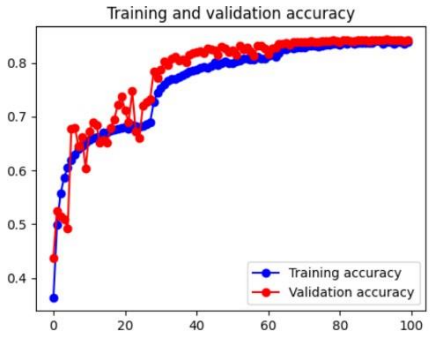
O reduce_lr, consiste no ajuste automático no *learning rate* durante o treino. O reduce_lr faz a monitorização de uma determinada métrica, no caso nós escolhemos o val_loss, porque achámos das melhores métricas para medir a aprendizagem durante o processo de treino, se o val_loss não melhorar, ou seja, diminuir durante 5 épocas, iremos ajustar o *learning rate* podendo atingir o LR até 0.000001.

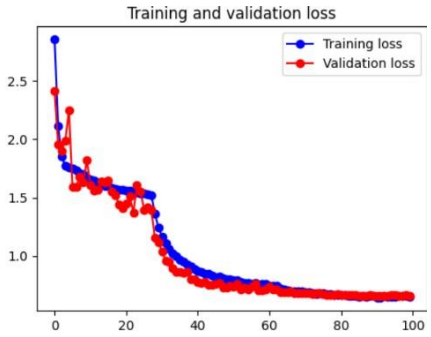
```python
# Callbacks
early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.1, patience=5, min_lr=0.000001)
```

Conclusão

No final, a nossa rede atingiu uma accuracy de 0.8395 e um val_acc de 0.8427, o que na nossa visão é um excelente resultado.

```
313/313 [==============================] - 4s 12ms/step - loss: 0.6615 - accuracy: 0.8427
val_acc: 0.8427000045776367
313/313 [==============================] - 4s 12ms/step - loss: 0.6814 - accuracy: 0.8372
Test accuracy: 0.8371999859809875
```





Os 10 treinos realizados foram

1º Treino

Aqui estava em utilização a arquitetura descrita acima

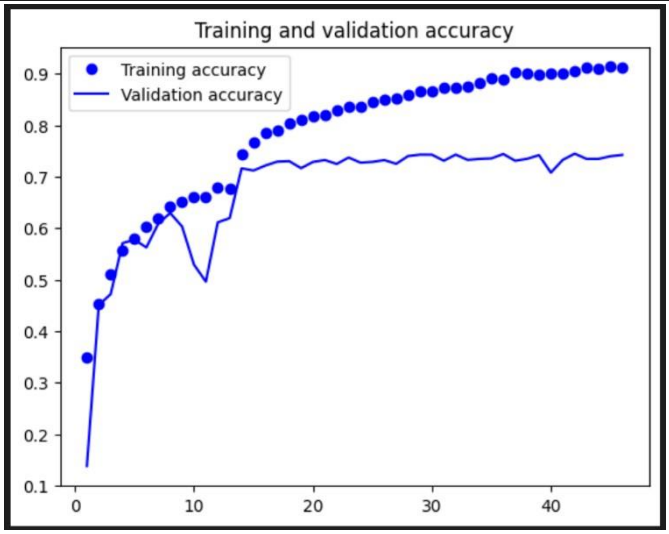
```
313/313 ━━━━━━━━━━━━━━━━━━━━ 93s 295ms/step - accuracy: 0.9783 - loss: 0.0685 - val_accuracy: 0.5917 - val_loss: 3.8113
Epoch 2/50
313/313 ━━━━━━━━━━━━━━━━━━━━ 91s 288ms/step - accuracy: 0.9891 - loss: 0.0330 - val_accuracy: 0.6033 - val_loss: 3.8288
Epoch 3/50
313/313 ━━━━━━━━━━━━━━━━━━━━ 90s 287ms/step - accuracy: 0.9897 - loss: 0.0315 - val_accuracy: 0.5782 - val_loss: 4.3744
Epoch 4/50
313/313 ━━━━━━━━━━━━━━━━━━━━ 90s 286ms/step - accuracy: 0.9928 - loss: 0.0249 - val_accuracy: 0.5761 - val_loss: 4.1851
Epoch 5/50
313/313 ━━━━━━━━━━━━━━━━━━━━ 90s 287ms/step - accuracy: 0.9820 - loss: 0.0712 - val_accuracy: 0.5892 - val_loss: 3.5818
Epoch 6/50
313/313 ━━━━━━━━━━━━━━━━━━━━ 90s 286ms/step - accuracy: 0.9944 - loss: 0.0164 - val_accuracy: 0.5940 - val_loss: 3.8864
Epoch 7/50
313/313 ━━━━━━━━━━━━━━━━━━━━ 90s 286ms/step - accuracy: 0.9989 - loss: 0.0043 - val_accuracy: 0.6044 - val_loss: 4.1234
Epoch 8/50
313/313 ━━━━━━━━━━━━━━━━━━━━ 90s 286ms/step - accuracy: 1.0000 - loss: 3.9251e-04 - val_accuracy: 0.6069 - val_loss: 4.1789
Epoch 9/50
313/313 ━━━━━━━━━━━━━━━━━━━━ 89s 282ms/step - accuracy: 1.0000 - loss: 5.5920e-05 - val_accuracy: 0.6083 - val_loss: 4.1987
Epoch 10/50
313/313 ━━━━━━━━━━━━━━━━━━━━ 90s 286ms/step - accuracy: 1.0000 - loss: 3.8960e-05 - val_accuracy: 0.6088 - val_loss: 4.2293
Epoch 11/50
313/313 ━━━━━━━━━━━━━━━━━━━━ 89s 284ms/step - accuracy: 1.0000 - loss: 2.9163e-05 - val_accuracy: 0.6094 - val_loss: 4.2614
Epoch 12/50
313/313 ━━━━━━━━━━━━━━━━━━━━ 89s 284ms/step - accuracy: 1.0000 - loss: 2.4703e-05 - val_accuracy: 0.6095 - val_loss: 4.2959
Epoch 13/50
313/313 ━━━━━━━━━━━━━━━━━━━━ 89s 284ms/step - accuracy: 1.0000 - loss: 2.1538e-05 - val_accuracy: 0.6101 - val_loss: 4.3233
...
313/313 ━━━━━━━━━━━━━━━━━━━━ 91s 289ms/step - accuracy: 1.0000 - loss: 6.3426e-06 - val_accuracy: 0.6110 - val_loss: 4.5737
Epoch 21/50
313/313 ━━━━━━━━━━━━━━━━━━━━ 94s 299ms/step - accuracy: 1.0000 - loss: 4.9775e-06 - val_accuracy: 0.6110 - val_loss: 4.6125
```

4 Treino

Do primeiro treino para o 4º treino não fizemos qualquer alteração de parâmetros ou hiperparâmetros pois estávamos a ver a rede a evoluir, no quarto treino a rede começou a parar de evoluir e tendo algum overfitting decidimos introduzir BatchNormalization e Dropout

```
model = Sequential()
model.add(Conv2D(32, (3, 3), padding='same', activation='relu', input_shape=(32, 32, 3)))
model.add(BatchNormalization())
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.25))

model.add(Conv2D(64, (3, 3), padding='same', activation='relu', kernel_regularizer=l2(0.001)))
model.add(BatchNormalization())
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.25))

model.add(Conv2D(128, (3, 3), padding='same', activation='relu', kernel_regularizer=l2(0.001)))
model.add(BatchNormalization())
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(512, activation='relu', kernel_regularizer=l2(0.001)))
model.add(BatchNormalization())
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))

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

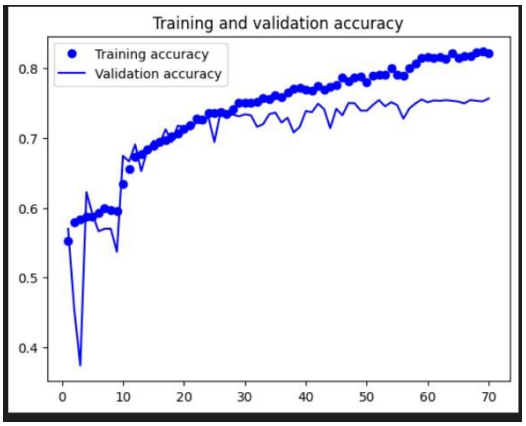
```
313/313 [==============================] - 8s 21ms/step - loss: 3.1427 - accuracy: 0.3482 - val_loss: 5.0835 - val_accuracy: 0.1382 - lr: 0.0010
Epoch 2/50
313/313 [==============================] - 6s 20ms/step - loss: 2.5576 - accuracy: 0.4520 - val_loss: 2.5139 - val_accuracy: 0.4507 - lr: 0.0010
Epoch 3/50
313/313 [==============================] - 6s 19ms/step - loss: 2.2575 - accuracy: 0.5102 - val_loss: 2.2980 - val_accuracy: 0.4719 - lr: 0.0010
Epoch 4/50
313/313 [==============================] - 6s 18ms/step - loss: 2.0337 - accuracy: 0.5572 - val_loss: 1.9539 - val_accuracy: 0.5709 - lr: 0.0010
Epoch 5/50
313/313 [==============================] - 7s 21ms/step - loss: 1.8790 - accuracy: 0.5807 - val_loss: 1.8758 - val_accuracy: 0.5774 - lr: 0.0010
Epoch 6/50
313/313 [==============================] - 7s 22ms/step - loss: 1.7735 - accuracy: 0.6040 - val_loss: 1.8800 - val_accuracy: 0.5626 - lr: 0.0010
Epoch 7/50
313/313 [==============================] - 7s 23ms/step - loss: 1.7155 - accuracy: 0.6187 - val_loss: 1.7653 - val_accuracy: 0.6087 - lr: 0.0010
Epoch 8/50
313/313 [==============================] - 7s 23ms/step - loss: 1.6485 - accuracy: 0.6423 - val_loss: 1.7066 - val_accuracy: 0.6294 - lr: 0.0010
Epoch 9/50
313/313 [==============================] - 6s 20ms/step - loss: 1.6388 - accuracy: 0.6515 - val_loss: 1.8330 - val_accuracy: 0.6032 - lr: 0.0010
Epoch 10/50
313/313 [==============================] - 7s 23ms/step - loss: 1.6287 - accuracy: 0.6615 - val_loss: 2.0391 - val_accuracy: 0.5292 - lr: 0.0010
Epoch 11/50
...
Epoch 45/50
313/313 [==============================] - 7s 22ms/step - loss: 0.6136 - accuracy: 0.9131 - val_loss: 1.2458 - val_accuracy: 0.7396 - lr: 1.0000e-04
Epoch 46/50
313/313 [==============================] - 7s 24ms/step - loss: 0.6112 - accuracy: 0.9114 - val_loss: 1.2131 - val_accuracy: 0.7422 - lr: 1.0000e-04
```



5 Treino

Do 4º para o 5º treino conseguimos baixar o *overfitting* aumentando o Dropout

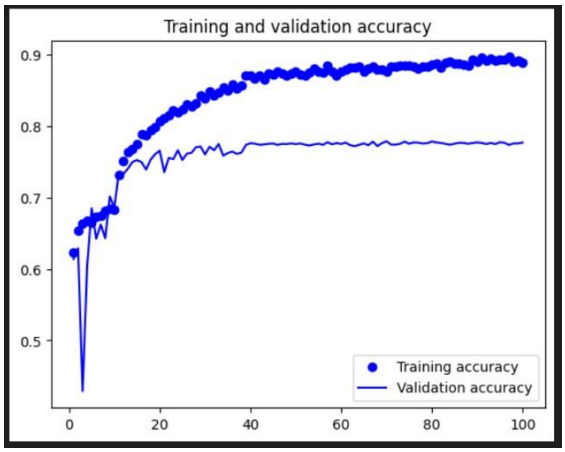
```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout, BatchNormalization
from tensorflow.keras.regularizers import l2
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau


num_classes = 10

model = Sequential()
model.add(Conv2D(32, (3, 3), padding='same', activation='relu', input_shape=(32, 32, 3)))
model.add(BatchNormalization())
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.4))

model.add(Conv2D(64, (3, 3), padding='same', activation='relu', kernel_regularizer=l2(0.01)))
model.add(BatchNormalization())
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.5))

model.add(Conv2D(128, (3, 3), padding='same', activation='relu', kernel_regularizer=l2(0.01)))
model.add(BatchNormalization())
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.5))

model.add(Flatten())
model.add(Dense(512, activation='relu', kernel_regularizer=l2(0.001)))
model.add(BatchNormalization())
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))

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



6 Treino

No sexto treino diminuímos os dropout's e aumentamos o número de épocas, mas embora conseguíssemos obter um melhor resultado verificamos que o overfitting começou a aumentar

```
Epoch 1/100
417/417 ━━━━━━━━━━━━━━━━━━━━ 9s 17ms/step - accuracy: 0.6555 - loss: 1.3017 - val_accuracy: 0.6140 - val_loss: 1.6994 - learning_rate: 0.0010
Epoch 2/100
417/417 ━━━━━━━━━━━━━━━━━━━━ 7s 16ms/step - accuracy: 0.6646 - loss: 1.5187 - val_accuracy: 0.6288 - val_loss: 1.7014 - learning_rate: 0.0010
Epoch 3/100
417/417 ━━━━━━━━━━━━━━━━━━━━ 8s 18ms/step - accuracy: 0.6735 - loss: 1.5429 - val_accuracy: 0.4291 - val_loss: 2.8255 - learning_rate: 0.0010
Epoch 4/100
417/417 ━━━━━━━━━━━━━━━━━━━━ 8s 19ms/step - accuracy: 0.6803 - loss: 1.5963 - val_accuracy: 0.6035 - val_loss: 1.9032 - learning_rate: 0.0010
Epoch 5/100
417/417 ━━━━━━━━━━━━━━━━━━━━ 8s 19ms/step - accuracy: 0.6623 - loss: 1.6594 - val_accuracy: 0.6849 - val_loss: 1.6429 - learning_rate: 0.0010
Epoch 6/100
417/417 ━━━━━━━━━━━━━━━━━━━━ 8s 19ms/step - accuracy: 0.6764 - loss: 1.6558 - val_accuracy: 0.6422 - val_loss: 1.8038 - learning_rate: 0.0010
Epoch 7/100
417/417 ━━━━━━━━━━━━━━━━━━━━ 8s 19ms/step - accuracy: 0.6848 - loss: 1.6661 - val_accuracy: 0.6621 - val_loss: 1.7670 - learning_rate: 0.0010
Epoch 8/100
417/417 ━━━━━━━━━━━━━━━━━━━━ 8s 19ms/step - accuracy: 0.6963 - loss: 1.6718 - val_accuracy: 0.6430 - val_loss: 1.8930 - learning_rate: 0.0010
Epoch 9/100
417/417 ━━━━━━━━━━━━━━━━━━━━ 7s 18ms/step - accuracy: 0.6817 - loss: 1.7185 - val_accuracy: 0.7014 - val_loss: 1.7024 - learning_rate: 0.0010
Epoch 10/100
417/417 ━━━━━━━━━━━━━━━━━━━━ 8s 19ms/step - accuracy: 0.6994 - loss: 1.6986 - val_accuracy: 0.6860 - val_loss: 1.7747 - learning_rate: 0.0010
Epoch 11/100
417/417 ━━━━━━━━━━━━━━━━━━━━ 8s 19ms/step - accuracy: 0.7331 - loss: 1.6248 - val_accuracy: 0.7351 - val_loss: 1.5929 - learning_rate: 1.0000e-04
Epoch 12/100
417/417 ━━━━━━━━━━━━━━━━━━━━ 8s 18ms/step - accuracy: 0.7451 - loss: 1.5362 - val_accuracy: 0.7330 - val_loss: 1.5640 - learning_rate: 1.0000e-04
Epoch 13/100
...
Epoch 99/100
417/417 ━━━━━━━━━━━━━━━━━━━━ 8s 20ms/step - accuracy: 0.8959 - loss: 0.6482 - val_accuracy: 0.7756 - val_loss: 1.0384 - learning_rate: 1.0000e-05
Epoch 100/100
417/417 ━━━━━━━━━━━━━━━━━━━━ 9s 21ms/step - accuracy: 0.8938 - loss: 0.6427 - val_accuracy: 0.7768 - val_loss: 1.0442 - learning_rate: 1.0000e-05
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

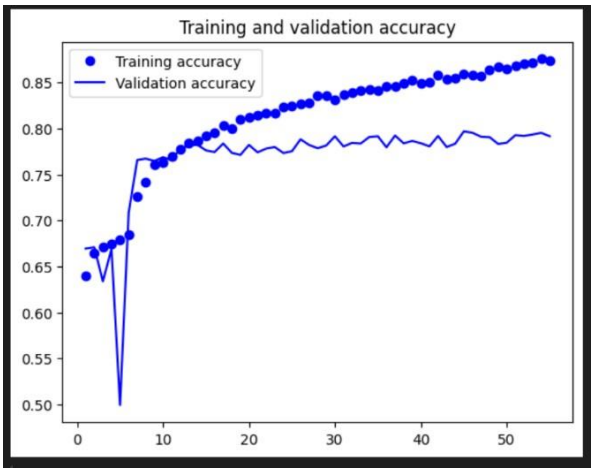


O 7 Treino não notamos qualquer melhoramento, por isso no 8 Treino voltamos a aumentar os dropouts e acrescentamos mais uma *Layer*

8 Treino

```
417/417 [==============================] - 4s 10ms/step - loss: 1.0124 - accuracy: 0.7968
val_acc: 0.7968000173568726
```

```
model = Sequential()
model.add(Conv2D(32, (3, 3), padding='same', activation='relu', input_shape=(32, 32, 3),kernel_regularizer=l2(0.001)))
model.add(BatchNormalization())
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.3))

model.add(Conv2D(64, (3, 3), padding='same', activation='relu', kernel_regularizer=l2(0.001)))
model.add(BatchNormalization())
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.4))

model.add(Conv2D(128, (3, 3), padding='same', activation='relu', kernel_regularizer=l2(0.001)))
model.add(BatchNormalization())
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.4))

model.add(Conv2D(256, (3, 3), padding='same', activation='relu', kernel_regularizer=l2(0.001)))
model.add(BatchNormalization())
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.4))

model.add(Flatten())
model.add(Dense(512, activation='relu', kernel_regularizer=l2(0.001)))
model.add(BatchNormalization())
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))

optimizer = Adam(learning_rate=0.001)
model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])
```



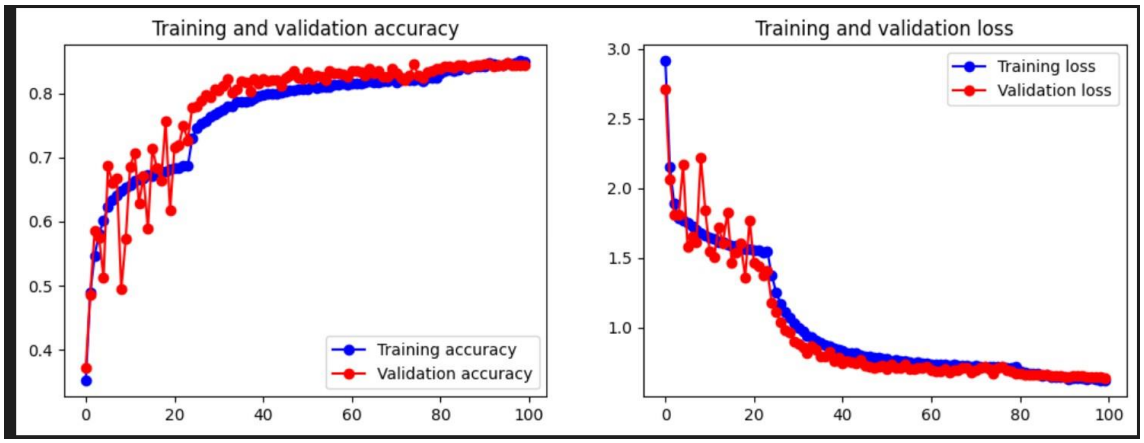
10º e último Treino

O 9º treino foi apenas uma tentativa falhada de reduzir o *overfitting*, por isso retornámos aos parâmetros da rede do 8º treino e em vez de 70 épocas utilizamos 100, implementámos callbacks (early_stopping e reduce_lr), o reduce_lr para tentarmos mitigar o overfitting e o early_stopping para nos ajudar a não ter de percorrer as 100 épocas se não estiver a haver uma evolução.

```
# Definindo o input
inputs = Input(shape=(32, 32, 3))

# Primeira camada convolucional
x = layers.Conv2D(32, (3, 3), activation='relu', padding='same', kernel_regularizer=l2(0.001))(inputs)
x = layers.BatchNormalization()(x)
x = layers.MaxPooling2D((2, 2))(x)
x = layers.Dropout(0.3)(x)

# Segunda camada convolucional
x = layers.Conv2D(64, (3, 3), activation='relu', padding='same', kernel_regularizer=l2(0.001))(x)
x = layers.BatchNormalization()(x)
x = layers.MaxPooling2D((2, 2))(x)
x = layers.Dropout(0.4)(x)

# Terceira camada convolucional
x = layers.Conv2D(128, (3, 3), activation='relu', padding='same', kernel_regularizer=l2(0.001))(x)
x = layers.BatchNormalization()(x)
x = layers.MaxPooling2D((2, 2))(x)
x = layers.Dropout(0.4)(x)

# Quarta camada convolucional
x = layers.Conv2D(256, (3, 3), activation='relu', padding='same', kernel_regularizer=l2(0.001))(x)
x = layers.BatchNormalization()(x)
x = layers.MaxPooling2D((2, 2))(x)
x = layers.Dropout(0.4)(x)

# Camada de Flatten
x = layers.Flatten()(x)

# Camada totalmente conectada
x = layers.Dense(512, activation='relu', kernel_regularizer=l2(0.001))(x)
x = layers.BatchNormalization()(x)
x = layers.Dropout(0.5)(x)

# Camada de saída
outputs = layers.Dense(10, activation='softmax')(x)  # Supondo 10 classes

# Definindo o modelo
model = Model(inputs=inputs, outputs=outputs)

# Compilando o modelo
optimizer = Adam(learning_rate=0.001)
model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])
```

```
1250/1250 [==============================] - 48s 37ms/step - loss: 2.9140 - accuracy: 0.3518 - val_loss: 2.7124 - val_accuracy: 0.3726 - lr: 0.0010
Epoch 2/100
1250/1250 [==============================] - 32s 26ms/step - loss: 2.1544 - accuracy: 0.4887 - val_loss: 2.0665 - val_accuracy: 0.4852 - lr: 0.0010
Epoch 3/100
1250/1250 [==============================] - 33s 26ms/step - loss: 1.8887 - accuracy: 0.5470 - val_loss: 1.8090 - val_accuracy: 0.5859 - lr: 0.0010
Epoch 4/100
1250/1250 [==============================] - 31s 25ms/step - loss: 1.7853 - accuracy: 0.5810 - val_loss: 1.8056 - val_accuracy: 0.5750 - lr: 0.0010
Epoch 5/100
1250/1250 [==============================] - 32s 25ms/step - loss: 1.7666 - accuracy: 0.6015 - val_loss: 2.1665 - val_accuracy: 0.5122 - lr: 0.0010
Epoch 6/100
1250/1250 [==============================] - 31s 25ms/step - loss: 1.7488 - accuracy: 0.6225 - val_loss: 1.5829 - val_accuracy: 0.6864 - lr: 0.0010
Epoch 7/100
1250/1250 [==============================] - 31s 25ms/step - loss: 1.7280 - accuracy: 0.6334 - val_loss: 1.6542 - val_accuracy: 0.6604 - lr: 0.0010
Epoch 8/100
1250/1250 [==============================] - 33s 26ms/step - loss: 1.7065 - accuracy: 0.6406 - val_loss: 1.6110 - val_accuracy: 0.6680 - lr: 0.0010
Epoch 9/100
1250/1250 [==============================] - 33s 27ms/step - loss: 1.6784 - accuracy: 0.6471 - val_loss: 2.2231 - val_accuracy: 0.4950 - lr: 0.0010
Epoch 10/100
1250/1250 [==============================] - 32s 26ms/step - loss: 1.6604 - accuracy: 0.6539 - val_loss: 1.8445 - val_accuracy: 0.5724 - lr: 0.0010
Epoch 11/100
...
Epoch 99/100
1250/1250 [==============================] - 33s 26ms/step - loss: 0.6195 - accuracy: 0.8503 - val_loss: 0.6478 - val_accuracy: 0.8431 - lr: 1.0000e-05
Epoch 100/100
1250/1250 [==============================] - 32s 26ms/step - loss: 0.6183 - accuracy: 0.8493 - val_loss: 0.6394 - val_accuracy: 0.8435 - lr: 1.0000e-05
```

```
313/313 [==============================] - 4s 12ms/step - loss: 0.6394 - accuracy: 0.8435
val_acc: 0.843500018119812
313/313 [==============================] - 4s 11ms/step - loss: 0.6599 - accuracy: 0.8439
Test accuracy: 0.8439000248908997
```



Para concluir, ao longo destes 10 treinos, apercebemo-nos que o Dropout é algo extremamente eficiente contra *overfitting*, quanto maior o número de épocas, maior vai ser a evolução da rede, e que um *learning rate* maior nem sempre é sinónimo de uma maior convergência.