Universidad de Castilla-La Mancha

Escuela Superior de Ingeniería Informática

Proyecto Final: Clasificación de Animales en una muestra desbalanceada

Grado en Ingeniería Informática

Visión Artifical y Reconocimiento de Patrones

Profesores

José Miguel Puerta Callejon

Luis González Naharro

Alumnos

Domingo José Caballero Navarro

Rubén Castillo Carrasco

Enero, 2023

Índice

Ilustraciones

1. Introducción

En primer lugar, llevaremos a cabo una explicación del problema que nos encontramos y cómo planificamos resolverlo. Además, debemos comentar que la elaboración de la libreta ha sido en Kaggle y se puede encontrar publicada en el siguiente enlace: https://www.kaggle.com/code/domingao/proyecto-varp.

El conjunto de datos seleccionado es una muestra de diez tipos de animales, pero dicha muestra se encuentra desbalanceada, esta particularidad hace que sea más interesante a la hora de abordar el problema dado que tendremos que elaborar una solución, bien realizando un Data Augmentation a las clases con menos instancias o bien entrenando nuestros modelos con otras métricas de rendimiento oportunas.

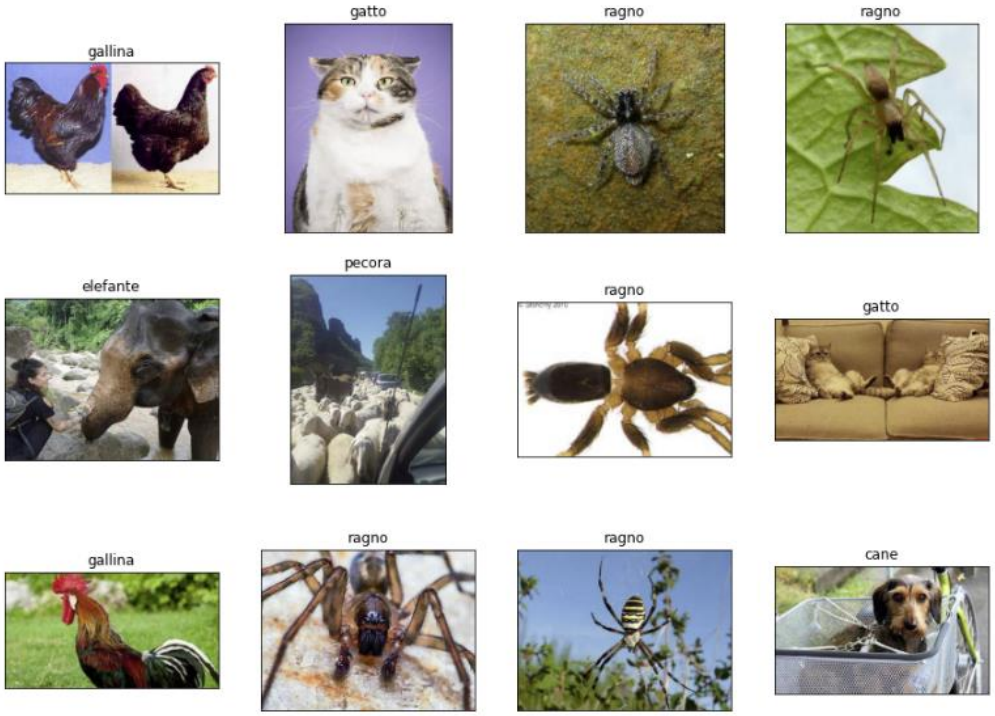

Ilustración 1: Muestra del conjunto de entrenamiento

Podemos ver la distribución de las clases si observamos el siguiente gráfico, comentar que dicho gráfico que se muestra está realizado sobre un conjunto de entrenamiento, el conjunto de prueba que ha sido preparado no se muestra en ningún momento para no ocasionar una fuga de datos.

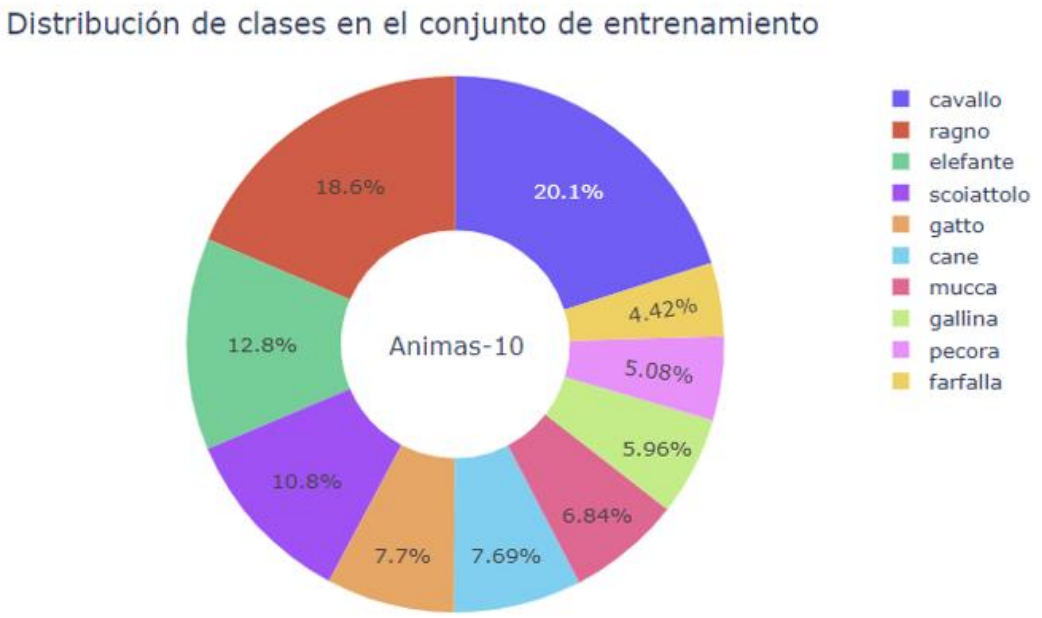

Ilustración 2: Distribución de las clases

Podemos apreciar cómo existen clases con una mayor y menor presencia en el conjunto de entrenamiento. Por ejemplo, la clase caballo aparece casi cinco veces más que la clase mariposa (farfalla).

2. Análisis Exploratorio de los Datos

Una vez conocemos el problema podemos empezar con el análisis de nuestro conjunto de datos, dado que tenemos que realizar el análisis sobre el conjunto de entrenamiento para no ocasionar fuga de datos en nuestros modelos deberemos realizar una separación en conjuntos de entrenamiento, validación y test.

Para realizar dicha división utilizamos split-folders, obteniendo así el conjunto de entrenamiento, validación y test. El conjunto de validación se utilizará para comparar los distintos modelos creados mientras que el de test para validar el modelo final. La división se realizaría de la siguiente forma:

```
splitfolders.ratio(dataset, output="output", seed=101, ratio=(.8, .1, .1))
```

Ilustración 3: División en entrenamiento, validación y test.

Donde dataset es la dirección donde se encuentra el conjunto de datos total. Una vez dividido entre los tres conjuntos de datos podemos empezar el análisis.

Mostraremos en primer lugar la distribución de la clase.

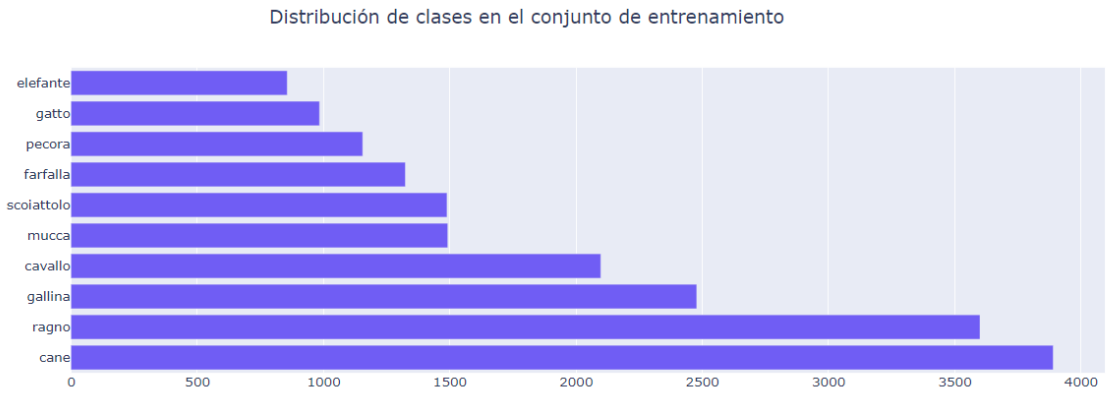

Ilustración 4: Distribución de las clases

Podemos observar varias cosas si analizamos la distribución de las clases, las tres clases mayoritarias ocupan la mitad del conjunto de datos, esto es un problema dado que nuestros modelos tenderán por predecir dichas clases. Además, observamos cómo las clases elefante y gato se encuentran por menos de 1000 instancias mientras que las dos mayoritarias superan las 3500 instancias, casi cuatro veces más.

Por lo tanto, una solución para este problema es realizar un Data Augmentation en función del número de instancias que hay en el conjunto de entrenamiento, es decir, aumentar la cantidad de instancias para las clases minoritarias llegando a balancear el conjunto de datos. Una segunda opción es realizar un Data Augmentation por igual a todas las clases y obligar a nuestro modelo a generalizar mejor sacando patrones diferenciales de cada clase. Para conseguir está segunda solución deberemos incluir a nuestros modelos ruido y Dropout, además, debemos cambiar la métrica de evaluación.

3. Preprocesamiento de los Datos

Una vez realizado en análisis y comentado el problema de tener una distribución de clases desbalanceada es hora de realizar el preprocesamiento. En primer lugar, optaremos por la segunda solución comentada y evaluaremos los modelos creados a partir de ella.

Por lo tanto, tendremos que realizar el Data Augmentation sobre todo el conjunto de entrenamiento. Para ello utilizaremos ImageDataGenerator de Tensorflow, lo que nos da la posibilidad de alterar las imágenes en cuanto a zoom, movimiento de la ventana, de translación, etc. Aplicaremos las siguientes alteraciones para el conjunto de entrenamiento:

```
train_datagen = ImageDataGenerator(zoom_range=0.15,
                                   width_shift_range=0.2,
                                   height_shift_range=0.2,
                                   shear_range=0.15)
```

Ilustración 5: Data Augmentation

Podemos apreciar cómo nuestro conjunto de entrenamiento se verá alterado en cuanto a zoom, movimiento de la ventana horizontalmente, movimiento de la ventana verticalmente y distorsión de la imagen sobre un eje dando otra perspectiva. Un ejemplo de esta alteración se vería de la siguiente forma:

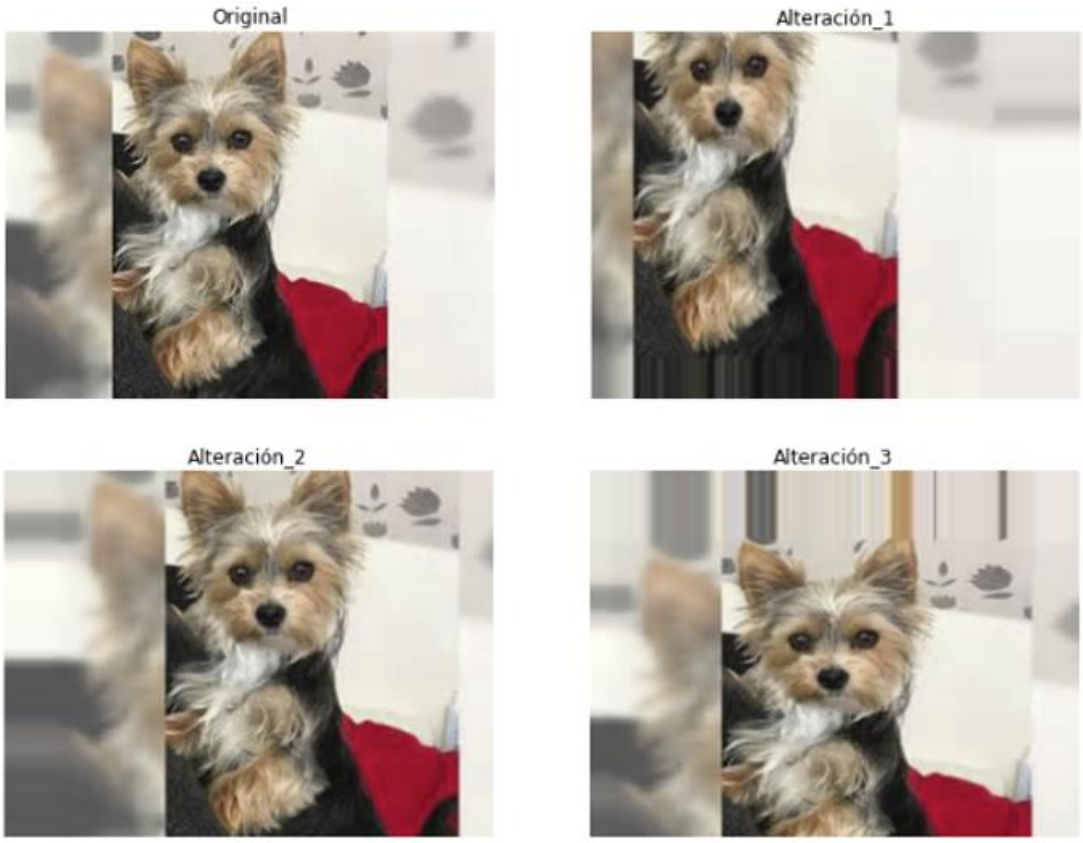

Ilustración 6: Ejemplo de alteración de imágenes

Como podemos observar hemos creado tres alteraciones de la imagen original para mostrar cómo funciona ImageDataGenerator, podemos apreciar cómo la imagen se desplaza tanto horizontal cómo verticalmente. Además, puede tener aplicado un zoom o se puede ver distorsionada en función a un eje. Por lo tanto, ya tenemos preparado cómo realizaremos un Data Augmentation sobre el conjunto de entrenamiento.

4. Diseño de diferentes modelos

Una vez realizado el Data Augmentation procedemos a diseñar los distintos modelos, tenemos que diferenciar dos grandes grupos que son los modelos creados manualmente a partir de redes convolucionales y los modelos preentrenados cargados a partir de Transfer Learning.

4.1. Diseño Redes Convolucionales

En este apartado diseñaremos manualmente una serie de redes convolucionales, para ello haremos uso de Tensorflow. Posteriormente compararemos los resultados obtenidos entre las distintas redes creadas. Distinguimos dos tipos de redes convolucionales:

4.1.1. Grandes Filtros

Encontramos redes convolucionales donde los filtros utilizados son grandes, en este caso elaboramos una red en la que el tamaño del kernel es de 11 o 13. Queremos comparar cómo afecta esto a la hora de clasificar cada tipo de animal y con que acierto lo hace.

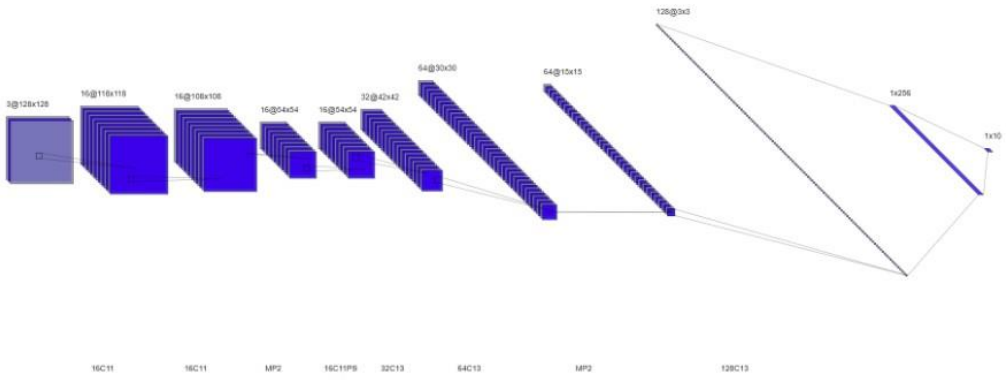

Ilustración 7: Red Convolucional con Grandes Filtros

Está sería una representación de cómo se perciben los filtros grandes. Para realizar dicha red convolucional utilizaremos las siguientes capas:

- **Conv2D**: encargada de aplicar la convolución a las imágenes.
- **BatchNormalization**: para aplicar una normalización cada vez que se aplica una convolución.
- **MaxPooling2D**: para reducir el tamaño de la imagen.
- **Dropout**: para obligar a generalizar al modelo.
- **Flatten**: para juntar las matrices de imágenes en un único array.
- **Dense**: para proporcionar una salida.

A continuación, mostraremos el orden de dichas capas en nuestro modelo:

```
model_cnn = keras.Sequential([
    keras.layers.Conv2D(filters = 16, kernel_size = 11, strides=1, activation=activation, input_shape=input_shape),
    keras.layers.BatchNormalization(),

    keras.layers.Conv2D(filters = 16, kernel_size = 11, strides=1, activation=activation),
    keras.layers.BatchNormalization(),

    keras.layers.MaxPooling2D(2),
    keras.layers.Dropout(0.2),

    keras.layers.Conv2D(filters = 16, kernel_size = 11, strides=1, padding="same", activation=activation),
    keras.layers.BatchNormalization(),

    keras.layers.Conv2D(filters = 32, kernel_size = 13, strides=1, activation=activation),
    keras.layers.BatchNormalization(),

    keras.layers.Conv2D(filters = 64, kernel_size = 13, strides=1, activation=activation),
    keras.layers.BatchNormalization(),

    keras.layers.MaxPooling2D(2),
    keras.layers.Dropout(0.2),

    keras.layers.Conv2D(filters = 128, kernel_size = 13, strides=1, activation=activation),
    keras.layers.BatchNormalization(),
    keras.layers.Dropout(0.2),

    keras.layers.Flatten(),
    keras.layers.Dense(256, activation=activation),
    keras.layers.Dropout(0.4),
    Dense(10, activation = "softmax")
])
```

Ilustración 8: CNN con Grandes Filtros

Podemos observar cómo hemos diseñado la red convolucional con grandes filtros, se trata de una red secuencial en la que encontramos las siguientes capas:

En primer lugar, encontramos capas convolucionales que son las capas Conv2D, dichas capas están seguidas por una capa BatchNormalization. En este caso encontramos 6 agrupaciones de este tipo a lo largo de nuestra red convolucional, pero lo más importante es destacar el tamaño del kernel en dichas capas. Nos encontramos ante un tamaño de kernel de 11 y 13, es decir haciendo que nuestra red tenga unos filtros grandes obligando a nuestra red a encontrar patrones diferenciales.

Tras aplicar unas series de convolucionales aparecen una capa MaxPooling2D, seguida de una capa Dropout, el objetivo de este tipo de capas es el de dotar a nuestro modelo de una mayor robustez obligando a generalizar mejor.

Por último, encontramos el grupo de capas encargadas de generar una salida, comienza con una capa Flatten encargada de aplanar los resultados de la capa anterior en una lista. Seguidamente aparece una capa oculta de tipo Dense y una capa Dropout, estas dos capas se encargan de obligar al modelo a generalizar y encontrar un patrón diferencial.

Por último, una última capa Dense se encarga de generar las probabilidades para cada clase utilizando la activación softmax.

A continuación, deberemos compilar nuestro modelo antes de empezar a entrenar. Para ello definimos las siguientes métricas para perdida, optimización y evaluación:

```
model_cnn.compile(loss = "categorical_crossentropy", optimizer = "adam", metrics = "AUC")
model_cnn.summary()
```

Ilustración 9: Perdida, optimizador y métrica

Como podemos observar la función de perdida viene definida por Categorical Crossentropy, calcula la perdida crossentropy entre las clases predichas y las verdaderas. En cuanto al optimizador el que se usa es el optimizador Adam y en cuanto a la métrica de evaluación es AUC, seleccionamos AUC dado que nuestro problema está desbalanceado.

Lo siguiente que realizaremos será establecer una serie de callbacks para ayudar a nuestro modelo a dos cosas: a conservar los mejores pesos de la mejor época del entrenamiento y a no entrenar en vano. Estas dos cosas las conseguimos con los siguientes callbacks:

```
cbs = [
    ModelCheckpoint(filepath = "modelo.h5",
                    monitor = "val_loss",
                    save_best_only = True,
                    save_weights_only=True),
    EarlyStopping(monitor='val_loss',
                  min_delta=0.0002,
                  patience=5)
]
```

Ilustración 10: Callbacks

Podemos apreciar cómo los callbacks que utilizamos son ModelCheckpoint, encargado de guardar los mejores pesos, y EarlyStopping, encargado de parar el entrenamiento si no hay mejora de perdida durante 5 epochs.

Una vez tenemos definido todo el modelo es momento de entrenarlo:

```
history_cnn = model_cnn.fit(train_generator,validation_data=val_generator,epochs=100,verbose=
1, callbacks=cbs)
```

Ilustración 11: Entrenamiento red CNN

Entrenamos el modelo con el conjunto de entrenamiento y lo validamos con el conjunto de validación. Entrenaremos durante 100 epochs dado que utilizaremos los callbacks explicados anteriormente, haciendo que el EarlyStopping detenga el entrenamiento cuando no exista mejora en los resultados de validación. El resultado de realizar dicho entrenamiento es el siguiente:

```
655/655 [==============================] - 286s 420ms/step - loss: 19.9186 - auc: 0.5640 - val_loss: 4.3230 - val_auc:
0.6058
Epoch 2/100
655/655 [==============================] - 275s 420ms/step - loss: 2.2178 - auc: 0.6351 - val_loss: 3.3841 - val_auc:
0.6478
Epoch 3/100
655/655 [==============================] - 278s 424ms/step - loss: 2.1914 - auc: 0.6424 - val_loss: 3.2959 - val_auc:
0.6385
Epoch 4/100
655/655 [==============================] - 281s 429ms/step - loss: 2.2171 - auc: 0.6375 - val_loss: 3.0489 - val_auc:
0.6445
Epoch 5/100
655/655 [==============================] - 278s 423ms/step - loss: 2.1961 - auc: 0.6396 - val_loss: 2.9499 - val_auc:
0.6470
Epoch 6/100
655/655 [==============================] - 283s 432ms/step - loss: 2.2018 - auc: 0.6376 - val_loss: 3.0580 - val_auc:
0.6524
Epoch 7/100
655/655 [==============================] - 285s 435ms/step - loss: 2.1869 - auc: 0.6404 - val_loss: 2.6172 - val_auc:
0.6450
Epoch 8/100
655/655 [==============================] - 278s 425ms/step - loss: 2.1881 - auc: 0.6399 - val_loss: 3.0163 - val_auc:
0.6514
Epoch 9/100
655/655 [==============================] - 281s 428ms/step - loss: 2.1814 - auc: 0.6425 - val_loss: 3.2252 - val_auc:
0.6424
Epoch 10/100
655/655 [==============================] - 281s 429ms/step - loss: 2.1823 - auc: 0.6424 - val_loss: 3.0806 - val_auc:
0.6082
Epoch 11/100
655/655 [==============================] - 275s 420ms/step - loss: 2.2111 - auc: 0.6370 - val_loss: 3.1108 - val_auc:
0.6307
Epoch 12/100
655/655 [==============================] - 276s 422ms/step - loss: 2.2079 - auc: 0.6305 - val_loss: 3.0287 - val_auc:
0.6306
```

Ilustración 12: Resultado entrenamiento red CNN

Si observamos cómo ha evolucionado el entrenamiento nos damos cuenta de que solo se ejecutan 12 epochs de las 100 que pusimos, esto sucede porque la perdida en la validación no mejora durante 5 epochs seguidas. Además, observamos que la mejor época viene dada por un 0.6524 en validación, un resultado muy bajo para lo esperado.

Podemos ver una representación gráfica de cómo ha ido evolucionando a continuación:

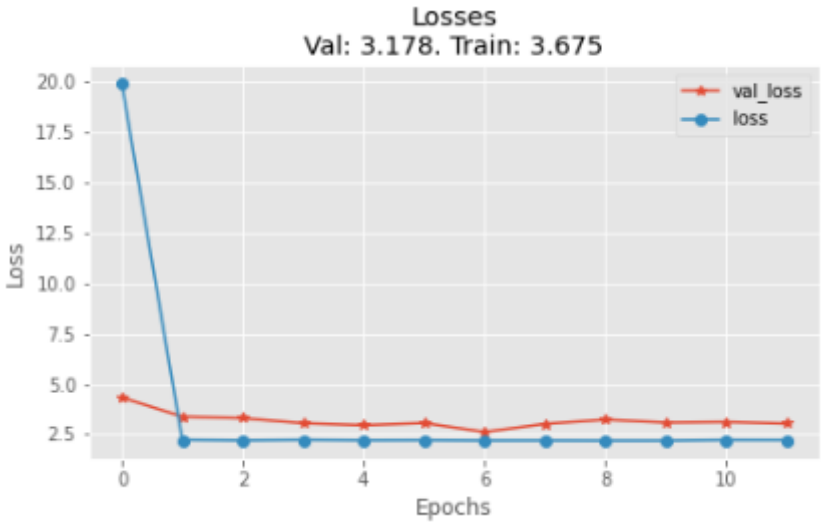

Ilustración 13: Resultado perdida red CNN

Si observamos bien la gráfica en la primera epoch existe una gran varianza entre el conjunto de entrenamiento y validación. Pero a partir de ese punto convergen al mismo error quedándose estancada la mejora.

A continuación, observaremos cómo ha ido evolucionando el resultado AUC en el conjunto de entrenamiento y de validación:

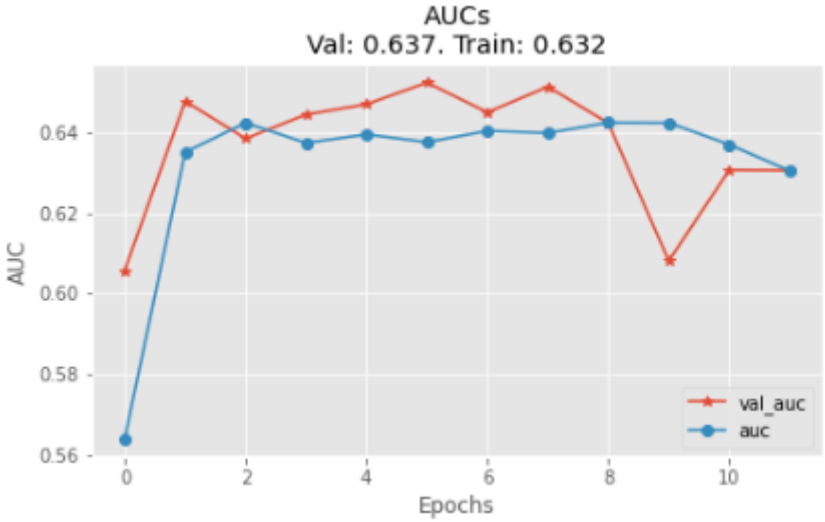

Ilustración 14: Resultado AUC red CNN

Podemos apreciar cómo existe esa diferencia al principio entre entrenamiento y validación que se mantiene, pero conforme pasan las épocas el resultado se estanca y no mejora, aunque sigue existiendo una mínima diferencia entre los conjuntos.

Algo a destacar es que se encuentra siempre por encima el conjunto de validación en cuanto a resultado, quizás esto viene ocasionado al entrenar con datos alterados en la fase Data Augmentation que hacen que evalué mejor los datos originales.

En conclusión, este modelo no es el óptimo y debemos probar otras alternativas que se acerquen a un resultado satisfactorio. Sin embargo, nos ha servido para comprender evaluar los diferentes modelos y que parámetros utilizar.

4.1.2. Pequeños Filtros

Hemos comprobado como la red convolucional con filtros grandes no es tan buena cómo esperábamos. Por lo tanto, probaremos con redes convolucionales donde los filtros utilizados son pequeños y la reducción de la imagen se realiza progresivamente. Para la elaboración de esta red convolucional utilizamos kernels de un menor tamaño, de 3, 5 y 7. Un ejemplo para entender mejor el funcionamiento de estos filtros es el siguiente:

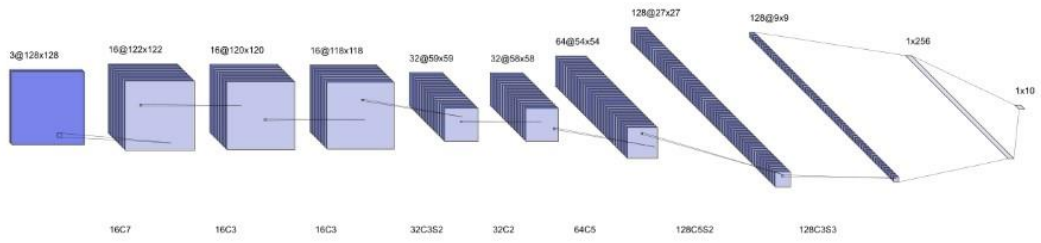

Ilustración 15: Red Convolucional con Pequeños Filtros

El concepto que hay detrás de utilizar filtros pequeños es el de alterar la imagen de una forma más pausada con el objetivo de comprobar que beneficia más a la red convolucional. Realizarlo el suavizado de las imágenes de una forma progresiva.

Para realizar dicha red convolucional utilizaremos las siguientes capas:

- **Conv2D**: encargada de aplicar la convolución a las imágenes.
- **BatchNormalization**: para aplicar una normalización cada vez que se aplica una convolución.
- **Dropout**: para obligar a generalizar al modelo.
- **Flatten**: para juntar las matrices de imágenes en un único array.
- **Dense**: para calcular el resultado en las capas ocultas y proporcionar una salida.

La ordenación de nuestro segundo modelo sería:

```
model_cnn_small = keras.Sequential([
    keras.layers.Conv2D(filters = 16, kernel_size = 7, strides=1, activation=activation, input_shape=input_shape),
    keras.layers.BatchNormalization(),

    keras.layers.Conv2D(filters = 16, kernel_size = 3, strides=1, activation=activation),
    keras.layers.BatchNormalization(),

    keras.layers.Conv2D(filters = 16, kernel_size = 3, strides=1, padding="same", activation=activation),
    keras.layers.BatchNormalization(),

    keras.layers.Conv2D(filters = 32, kernel_size = 3, strides=1, activation=activation),
    keras.layers.BatchNormalization(),

    keras.layers.Conv2D(filters = 32, kernel_size = 3, strides=1, activation=activation),
    keras.layers.BatchNormalization(),

    keras.layers.Conv2D(filters = 64, kernel_size = 5, strides=1, activation=activation),
    keras.layers.BatchNormalization(),

    keras.layers.Conv2D(filters = 128, kernel_size = 5, strides=3, activation=activation),
    keras.layers.BatchNormalization(),

    keras.layers.Conv2D(filters = 128, kernel_size = 3, strides=3, activation=activation),
    keras.layers.BatchNormalization(),

    keras.layers.Dropout(0.2),

    keras.layers.Flatten(),
    keras.layers.Dense(256, activation=activation),
    keras.layers.Dropout(0.4),
    Dense(10, activation = "softmax")
])
```

Ilustración 16: CNN con Pequeños Filtros

Como podemos observar en este modelo hemos prescindido también de las capas MaxPooling2D con el objetivo de evaluar si el hecho de reducir las imágenes afectaba a la perdida de información.

En cuanto al diseño de la red convolucional esta compuesta por 8 capas Conv2D seguidas de capas BatchNormalization. Una vez se han aplicado esas capas se aplana el resultado con una capa Flatten y se calcula un resultado con las capas Dense.

Lo siguiente que haremos una vez tenemos diseñado nuestro modelo es compilarlo utilizando las mismas métricas seleccionadas anteriormente.

```
model_cnn_small.compile(loss = "categorical_crossentropy", optimizer = "adam", metrics = "AUC")
model_cnn_small.summary()
```

Ilustración 17: Perdida, optimizador y métrica

Una vez tenemos compilado nuestro modelo es hora de realizar el entrenamiento de la misma forma que antes, durante 100 epochs dado que tenemos callbacks:

```
history_cnn_small = model_cnn_small.fit(train_generator,validation_data=val_generator,epochs=100,verbose=1, callbacks=cbs)
```

Ilustración 18: Entrenamiento red CNN filtros pequeños

El resultado del entrenamiento sería el siguiente:

```
Epoch 1/100
655/655 [==============================] - 301s 454ms/step - loss: 3.4088 - auc: 0.6063 - val_loss: 2.2110 - val_auc:
0.6815
Epoch 2/100
655/655 [==============================] - 300s 458ms/step - loss: 2.1518 - auc: 0.6740 - val_loss: 2.5128 - val_auc:
0.6970
Epoch 3/100
655/655 [==============================] - 297s 454ms/step - loss: 2.0831 - auc: 0.7056 - val_loss: 1.9901 - val_auc:
0.7691
Epoch 4/100
655/655 [==============================] - 300s 458ms/step - loss: 1.9894 - auc: 0.7427 - val_loss: 2.1194 - val_auc:
0.7402
Epoch 5/100
655/655 [==============================] - 294s 449ms/step - loss: 1.9308 - auc: 0.7616 - val_loss: 1.8476 - val_auc:
0.7924
Epoch 6/100
655/655 [==============================] - 295s 450ms/step - loss: 1.8949 - auc: 0.7721 - val_loss: 1.9656 - val_auc:
0.7638
Epoch 7/100
655/655 [==============================] - 295s 450ms/step - loss: 1.8704 - auc: 0.7804 - val_loss: 2.5253 - val_auc:
0.7167
Epoch 8/100
655/655 [==============================] - 296s 452ms/step - loss: 1.8339 - auc: 0.7907 - val_loss: 1.6130 - val_auc:
0.8511
Epoch 9/100
655/655 [==============================] - 294s 449ms/step - loss: 1.8030 - auc: 0.7974 - val_loss: 1.8459 - val_auc:
0.8029
Epoch 10/100
655/655 [==============================] - 295s 450ms/step - loss: 1.7739 - auc: 0.8060 - val_loss: 1.5613 - val_auc:
0.8682
Epoch 11/100
655/655 [==============================] - 296s 452ms/step - loss: 1.7294 - auc: 0.8163 - val_loss: 1.5557 - val_auc:
0.8599
Epoch 12/100
655/655 [==============================] - 296s 452ms/step - loss: 1.7041 - auc: 0.8238 - val_loss: 1.7581 - val_auc:
0.8235
Epoch 13/100
655/655 [==============================] - 295s 450ms/step - loss: 1.6916 - auc: 0.8256 - val_loss: 1.4732 - val_auc:
0.8738
Epoch 14/100
655/655 [==============================] - 298s 454ms/step - loss: 1.6588 - auc: 0.8348 - val_loss: 1.4162 - val_auc:
0.8843
Epoch 15/100
655/655 [==============================] - 299s 456ms/step - loss: 1.6361 - auc: 0.8399 - val_loss: 1.4291 - val_auc:
0.8853
Epoch 16/100
655/655 [==============================] - 299s 457ms/step - loss: 1.6365 - auc: 0.8398 - val_loss: 1.3941 - val_auc:
0.8897
Epoch 17/100
655/655 [==============================] - 297s 453ms/step - loss: 1.5983 - auc: 0.8486 - val_loss: 1.3970 - val_auc:
0.8880
Epoch 18/100
655/655 [==============================] - 300s 458ms/step - loss: 1.5941 - auc: 0.8503 - val_loss: 1.4947 - val_auc:
0.8763
Epoch 19/100
655/655 [==============================] - 301s 459ms/step - loss: 1.5649 - auc: 0.8554 - val_loss: 1.8505 - val_auc:
0.8349
Epoch 20/100
655/655 [==============================] - 301s 460ms/step - loss: 1.5506 - auc: 0.8577 - val_loss: 1.5671 - val_auc:
0.8642
Epoch 21/100
655/655 [==============================] - 299s 456ms/step - loss: 1.5271 - auc: 0.8633 - val_loss: 1.7919 - val_auc:
0.8786
```

Ilustración 19: Entrenamiento CNN con filtros pequeños

Como se puede observar en la ilustración 19 el entrenamiento ha sido mucho más duradero que en el caso de la primera red convolucional creada. Además, ha ofrecido un mejor resultado siendo la mejor época un 0.8897.

Si observamos cómo ha evolucionado la perdida sobre el conjunto de validación y entrenamiento veríamos lo siguiente:

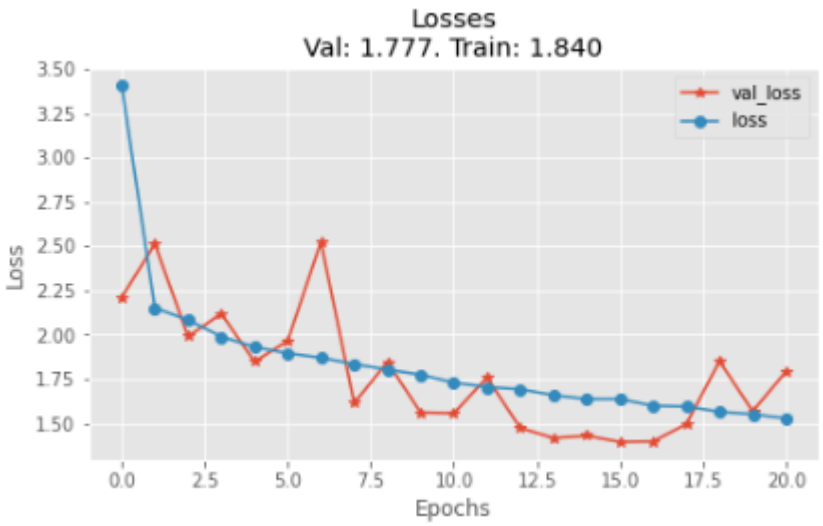

Ilustración 20: Perdida CNN con filtros pequeños

Podemos apreciar una pequeña varianza entre el conjunto de entrenamiento y validación que se mantiene durante todo el entrenamiento. En cuanto a la evolución del sesgo este se va reduciendo conforme avanza el entrenamiento. Si observamos la gráfica para la métrica de evaluación tendría la siguiente apariencia:

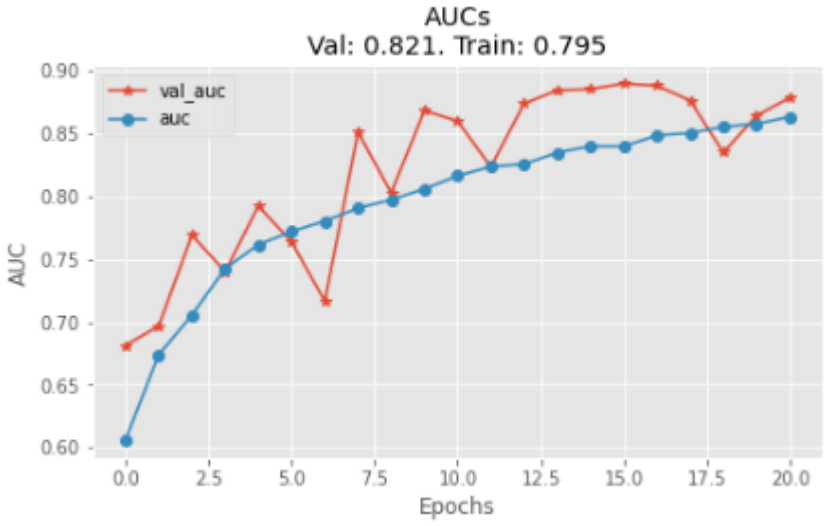

Ilustración 21: AUC de CNN con filtros pequeños

Se puede apreciar al igual que en la gráfica de perdida cómo existe esa varianza entre el conjunto de validación y de entrenamiento. En conclusión, obtenemos un buen resultado en el conjunto de validación, pero creemos que está muy lejos de ser una solución prometedora. Por lo tanto, tendremos que buscar otros diseños con el objetivo de encontrar el modelo que nos ofrezca un resultado óptimo.

4.2. Diseño Redes de Transfer Learning

En este apartado realizaremos la implementación de redes neuronales profundas preentrenadas, comparándolas así con las redes convolucionales previamente creadas por nosotros.

El objetivo de utilizar dichas redes preentrenadas es aprovechar las principales ventajas que ofrecen, como puede ser unos pesos óptimos ya establecidos. Las redes que hemos decido utilizar son las siguientes:

4.2.1. ResNet50

La primera red preentrenada que implementaremos es la red ResNet50, es una red residual lo que quiere decir que las capas intermedias pueden estar enlazadas con capas no contiguas. Para implementar dicha red debemos cargarla desde el módulo de applications de Keras.

```
from tensorflow.keras.applications import ResNet50

model_resnet50 = ResNet50(
      input_shape = (224,224,3),
      include_top = False,
      weights = 'imagenet'
    )
```

Ilustración 22: Carga de ResNet50

Una vez tenemos cargado nuestra red preentrenada lo que haremos será congelar las capas intermedias para que mantenga los pesos ya calculados.

```
for layers in model_resnet50.layers:
    layers.trainable = False
```

Ilustración 23: Congelar Capas

Lo siguiente que haremos será proporcionar una salida para la clasificación, para generar dicha salida tendremos que conectar la salida del modelo con unas capas Dense.

```
y = Flatten()(model_resnet50.output)
y = Dropout(0.5)(y)
y = Dense(10, activation = "softmax")(y)

model_resnet50 = keras.Model(model_resnet50.input, y)
model_resnet50.compile(loss = "categorical_crossentropy", optimizer = "adam", metrics = "AUC")
model_resnet50.summary()
```

Ilustración 24: Salida ResNet50

Una vez tenemos cómo se va a generar la salida debemos compilar nuestro modelo, cómo se puede apreciar en la ilustración 24 las métricas seleccionadas son las mismas que en los modelos creados anteriormente.

El entrenamiento de nuestro modelo con una red preentrenada se realizará de igual forma que en los modelos anteriores, entrenaremos durante 100 epochs dado que contamos con el callback EarlyStopping que detendrá el entrenamiento cuando no mejore. El resultado del entrenamiento es el siguiente:

```
Epoch 1/100
655/655 [==============================] - 280s 423ms/step - loss: 4.8333 - auc: 0.9184 - v
al_loss: 3.4369 - val_auc: 0.9528
Epoch 2/100
655/655 [==============================] - 277s 422ms/step - loss: 5.0587 - auc: 0.9399 - v
al_loss: 4.1397 - val_auc: 0.9593
Epoch 3/100
655/655 [==============================] - 276s 421ms/step - loss: 5.0243 - auc: 0.9471 - v
al_loss: 4.7457 - val_auc: 0.9589
Epoch 4/100
655/655 [==============================] - 277s 424ms/step - loss: 5.1185 - auc: 0.9521 - v
al_loss: 4.9728 - val_auc: 0.9624
Epoch 5/100
655/655 [==============================] - 276s 422ms/step - loss: 4.8672 - auc: 0.9562 - v
al_loss: 5.4058 - val_auc: 0.9630
Epoch 6/100
655/655 [==============================] - 275s 420ms/step - loss: 4.7443 - auc: 0.9579 - v
al_loss: 6.0654 - val_auc: 0.9601
```

Ilustración 25: Entrenamiento ResNet50

Podemos observar cómo el entrenamiento se detiene en la sexta época, es decir no ha conseguido mejorar la perdida en ninguna epoch. En cuanto al resultado obtenido en validación es muy bueno, el mejor resultado es de 0.9630.

Si observamos cómo ha evolucionado la perdida en el conjunto de validación y entrenamiento:

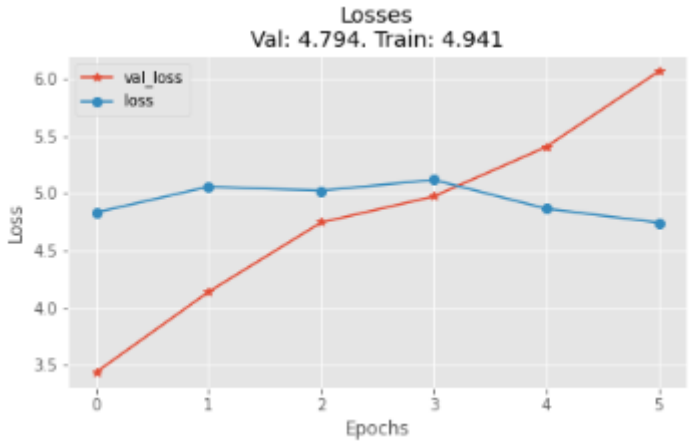

Ilustración 26: Perdida red ResNet50

Encontramos cómo existe una pequeña varianza entre el conjunto de entrenamiento y entre el conjunto de validación. Además, encontramos un poco de sesgo dado que el error está un pelín por encima del óptimo. En cuanto a la evolución de la métrica de evaluación obtenemos la siguiente gráfica:

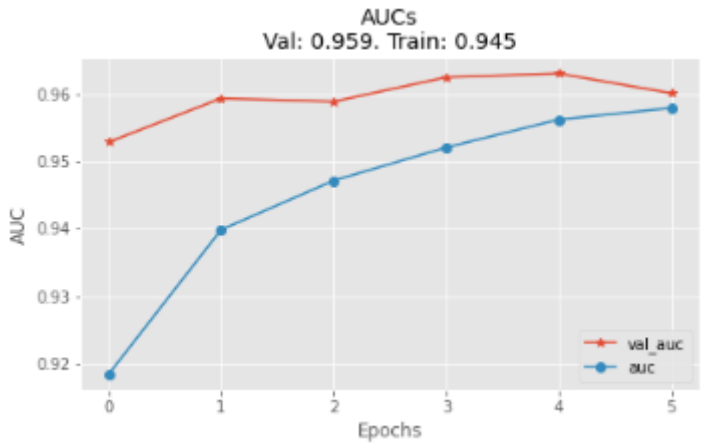

Ilustración 27: AUC red ResNet50

Podemos confirmar dicha varianza comentada en la gráfica de perdida, en este caso también la encontramos. Aunque mencionar que en las últimas épocas esa varianza va desapareciendo y se encuentran a una menor distancia.

En conclusión, obtenemos un modelo muy bueno en cuanto ha resultado obtenido y por lo tanto deberemos tenerlo en cuenta a la hora de seleccionar un modelo final comparándolo con el resto de los modelos creados.

4.2.2. EfficientNetB7

Viendo la buena aproximación de la red preentrenada seguimos implementado dichas redes. Por lo tanto, la siguiente red que utilizaremos es la red EfficientNetB7, tendremos que seguir los mismos pasos que con la red ResNet50 y el primero será cargar el modelo del módulo applications de Keras:

```
from tensorflow.keras.applications import EfficientNetB7

model_efficientnetb7 = EfficientNetB7(
      input_shape = (224,224,3),
      include_top = False,
      weights = 'imagenet'
    )
```

Ilustración 28: Cargamos EfficientNetB7

Al igual que hicimos con la red ResNet50 haremos con la EfficientNetB7, por lo tanto, congelaremos las capas intermedias del modelo. Después de hacerlo crearemos la salida de nuestra red con una capa Dense:

```
y = Flatten()(model_efficientnetb7.output)
y = Dropout(0.5)(y)
y = Dense(10, activation = "softmax")(y)

model_efficientnetb7 = keras.Model(model_efficientnetb7.input, y)
model_efficientnetb7.compile(loss = "categorical_crossentropy", optimizer = "adam", metrics =
"AUC")
model_efficientnetb7.summary()
```

Ilustración 29: Salida EfficientNetB7

Como podemos observar el diseño que ofrece la salida en la clasificación es igual que en la red ResNet50. Además, también podemos observar cómo la compilación se lleva a cabo de igual forma que llevamos haciendo en todas las redes creadas.

El entrenamiento que se lleva a cabo es de igual forma, se establecen 100 epochs y se espera que el callback EarlyStopping detenga el entrenamiento cuando no mejore. El resultado es el siguiente:

```
Epoch 1/100
655/655 [==============================] - 377s 549ms/step - loss: 1.3871 - auc: 0.9667 - v
al_loss: 1.1296 - val_auc: 0.9807
Epoch 2/100
655/655 [==============================] - 342s 522ms/step - loss: 1.3066 - auc: 0.9750 - v
al_loss: 1.5293 - val_auc: 0.9777
Epoch 3/100
655/655 [==============================] - 341s 520ms/step - loss: 1.2885 - auc: 0.9777 - v
al_loss: 1.4032 - val_auc: 0.9818
Epoch 4/100
655/655 [==============================] - 348s 530ms/step - loss: 1.2191 - auc: 0.9799 - v
al_loss: 1.9259 - val_auc: 0.9767
Epoch 5/100
655/655 [==============================] - 359s 547ms/step - loss: 1.2676 - auc: 0.9812 - v
al_loss: 1.6012 - val_auc: 0.9807
Epoch 6/100
655/655 [==============================] - 349s 533ms/step - loss: 1.2055 - auc: 0.9821 - v
al_loss: 1.6671 - val_auc: 0.9835
```

Ilustración 30: Entrenamiento EfficientNetB7

Podemos observar cómo sucede lo mismo que en el modelo anterior, el entrenamiento se detiene a las 6 epochs dado que no consigue mejorar la perdida. Si nos fijamos en el resultado de validación nos encontraremos con un muy buen resultado, es un 0.9835. Sin duda es una mejora respecto de los anteriores modelos muy buena. A continuación, visualizaremos la gráfica correspondiente a la perdida:

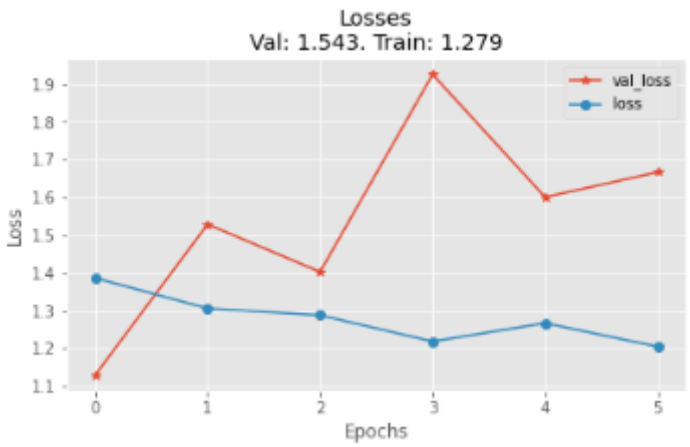

Ilustración 31: Perdida EfficientNetB7

Podemos observar cómo existe una pequeña varianza entre el conjunto de validación y entrenamiento. En cuanto al sesgo es muy bueno dado que el error es menor de 2 siendo muy cercano al óptimo. Si observamos cómo ha ido mejorando el AUC durante el entrenamiento encontramos la siguiente gráfica:

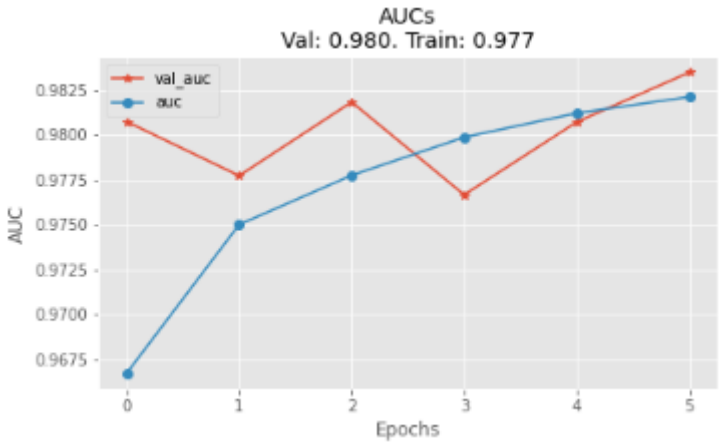

Ilustración 32: AUC EfficientNetB7

Existe una mínima varianza entre los conjuntos de datos, pero acaban convergiendo hacia un mismo punto minimizan esa pequeña varianza. En conclusión, nos encontramos ante el mejor modelo que hemos conseguido hasta el momento siendo una solución muy prometedora para nuestro problema.

4.2.3. EfficientNetB5

Viendo el sorprendente resultado que hemos obtenido con la red EfficientNetB7 probaremos con una de sus hermanas destacadas cómo es EfficientNetB5. Por lo tanto, seguiremos el mismo proceso cargando primeramente la red:

```
from tensorflow.keras.applications import EfficientNetB5

model_efficientnetb5 = EfficientNetB5(
      input_shape = (224,224,3),
      include_top = False,
      weights = 'imagenet'
    )
```

Ilustración 33: Carga de EfficientNetB5

Dado que seguirmos los mismos pasos que con la red EfficientNetB7 lo primero que haremos será congelar las capas intermedias del modelo. Después de hacerlo crearemos la salida de nuestra red con una capa Dense por igual.

```
y = Flatten()(model_efficientnetb5.output)
y = Dropout(0.5)(y)
y = Dense(10, activation = "softmax")(y)

model_efficientnetb5 = keras.Model(model_efficientnetb5.input, y)
model_efficientnetb5.compile(loss = "categorical_crossentropy", optimizer = "adam", metrics =
"AUC")
model_efficientnetb5.summary()
```

Ilustración 34: Salida EfficientNetB5

Como hemos comentado el diseño será igual que para EfficientNetB7. Además, también podemos observar cómo la compilación se lleva a cabo de igual forma que llevamos haciendo en todas las redes creadas.

El entrenamiento que se lleva a cabo es de igual forma, se establecen 100 epochs y se espera que el callback EarlyStopping detenga el entrenamiento cuando no mejore. El resultado es el siguiente:

```
Epoch 1/100
655/655 [==============================] - 325s 479ms/step - loss: 1.3173 - auc: 0.9649 - v
al_loss: 1.1427 - val_auc: 0.9769
Epoch 2/100
655/655 [==============================] - 307s 469ms/step - loss: 1.2301 - auc: 0.9749 - v
al_loss: 1.2966 - val_auc: 0.9797
Epoch 3/100
655/655 [==============================] - 307s 469ms/step - loss: 1.2218 - auc: 0.9774 - v
al_loss: 1.4505 - val_auc: 0.9784
Epoch 4/100
655/655 [==============================] - 309s 471ms/step - loss: 1.1726 - auc: 0.9790 - v
al_loss: 1.6228 - val_auc: 0.9796
Epoch 5/100
655/655 [==============================] - 301s 459ms/step - loss: 1.0525 - auc: 0.9819 - v
al_loss: 1.6573 - val_auc: 0.9811
Epoch 6/100
655/655 [==============================] - 315s 481ms/step - loss: 1.2351 - auc: 0.9798 - v
al_loss: 1.7942 - val_auc: 0.9815
```

Ilustración 35: Entrenamiento EfficientNetB5

Observamos como sucede de igual manera que ha pasado en las dos redes preentrenadas ResNet50 y EfficientNetB7, con sólo 6 epochs es suficiente dado que no consigue una mejora en la perdida sobre el conjunto de validación. En cuanto al resultado obtenido por este modelo encontramos que el mejor resultado es un 0.9815, sin duda siendo un muy buen resultado parecido al de EfficientNetB7.

Si observamos cómo evoluciona la pérdida de una forma gráfica veremos lo siguiente:

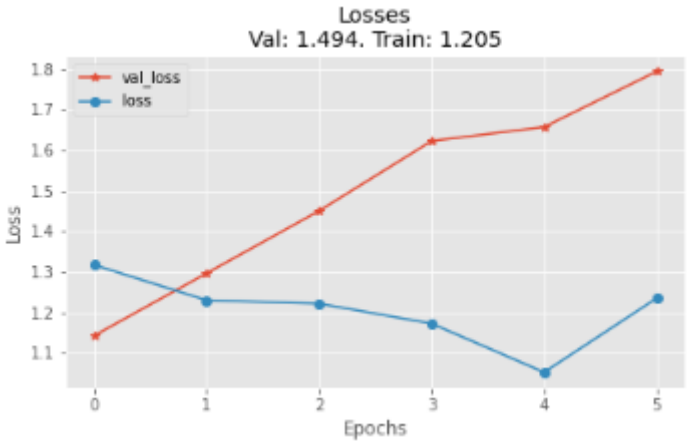

Ilustración 36: Perdida EfficientNetB5

Podemos destacar una pequeña varianza entre el conjunto de validación y de entrenamiento, aunque en la gráfica se vea abultada es pequeña dado los valores que toma el eje y de la gráfica. Por otro lado, encontramos un pelín de sesgo, pero este sigue siendo ínfimo y cercano al óptimo. Si visualizamos el AUC de nuestro modelo:

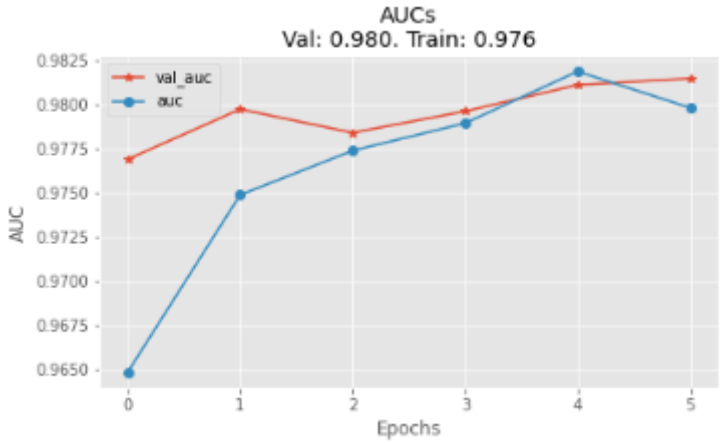

Ilustración 37: AUC EfficientNetB5

En este caso observamos esa pequeña diferencia de AUC durante las primeras épocas, pero conforme avanzan el entrenamiento el AUC tanto del conjunto de entrenamiento como el de validación converge hacia un mismo punto.

En conclusión, al igual que sucedía con EfficientNetB7 estamos ante un gran modelo y muy prometedor para nuestro problema. Por lo tanto, llegados a este punto no diseñaremos más modelos dado que tenemos tres modelos muy buenos que pueden ser la solución final.

5. Análisis de los resultados obtenidos

Los tres mejores modelos provienen de las redes preentrenadas, llegados a este punto es hora de realizar una comparación entre dichos modelos y comprobar que resultado nos ofrecen. Para ello enfrentaremos dichos modelos frente al test y compararemos los resultados obtenidos.

Primero de todo, hay que comentar que en la libreta se utiliza una carpeta que hemos diseñado en la que almacenamos los modelos y posteriormente son cargados, esto lo realizamos dado que la larga duración de los modelos no nos permitía realizar pruebas rápido y teníamos que aprovechar el tiempo que nos ofrecía Kaggle de GPU.

Una vez comentado esto procedemos a evaluar los modelos frente al test de la siguiente forma:

```
test_loss_resnet50, test_auc_resnet50 = model_resnet50.evaluate(test_generator, verbose=1)
test_loss_efficientnetb7, test_auc_efficientnetb7 = model_efficientnetb7.evaluate(test_generat
or, verbose=1)
test_loss_efficientnetb5, test_auc_efficientnetb5 = model_efficientnetb5.evaluate(test_generat
or, verbose=1)
```

Ilustración 38: Evaluación Modelos

Ofreciendo los siguientes resultados en el mismo orden que aparecen en la ilustración 38:

```
83/83 [==============================] - 10s 103ms/step - loss: 5.3336 - auc: 0.9634
83/83 [==============================] - 24s 235ms/step - loss: 2.0402 - auc: 0.9789
83/83 [==============================] - 16s 143ms/step - loss: 1.9561 - auc: 0.9757
```

Ilustración 39: Resultados Evaluación Modelos

Si observamos los resultados el mejor modelo que obtenemos en cuanto a AUC es el modelo EfficientNetB7, en cuanto a pérdida es el modelo EfficientNetB5. Podemos observar cómo los modelos de EfficientNet son muy buenos para este tipo de problemas siendo muy sencillos de utilizar.

Podemos ordenar los modelos según tres valoraciones posibles que son tiempo de inferencia, perdida y AUC:

	Inferencia	**Perdida**	**AUC**
Primero	ResNet50	EfficientNetB5	EfficientNetB7
Segundo	EfficientNetB5	EfficientNetB7	EfficientNetB5
Tercero	EfficientNetB7	ResNet50	ResNet50

Ilustración 40: Tabla de resultados

Interpretando los datos obtenidos y visualizando la tabla podemos destacar dos cosas acerca de los resultados:

- Quizás la prioridad de ResNet50 por ofrecer una inferencia más rápida repercuta en los resultados obtenidos.
- En cuanto a relación AUC y tiempo de inferencia EfficientNetB5 ofrece un mejor resultado.

Los tres modelos son muy buenos, pero quizás esa pequeña perdida que hemos notado en el resultado obtenido por el test no sea lo mejor dado que hemos tenido un 0.9789 cuando nuestros modelos tenían un resultado superior a 0.98 en el conjunto de validación.

6. Modelo Final: Ensemble

Debido al problema comentado anteriormente, el deterioro de resultado AUC entre el conjunto de validación y el conjunto final de test creemos conveniente resolver ese problema para optimizar el resultado en la clasificación. Para solventar dicho problema hemos decido construir un ensemble con los tres mejores modelos.

El ensemble que se realizará será el más simple posible para ver si tiene un efecto positivo en el resultado, vendría a ser una media de las probabilidades. Es decir, cada modelo predice un array con las probabilidades para cada clase, por lo tanto, lo que haremos será sumar los arrays para la misma imagen obteniendo la suma de los tres modelos. Una vez tenemos la suma de los tres modelos procedemos a sacar la máxima probabilidad de cada imagen obteniendo el resultado final.

Una representación de dicho ensemble podría ser la siguiente:

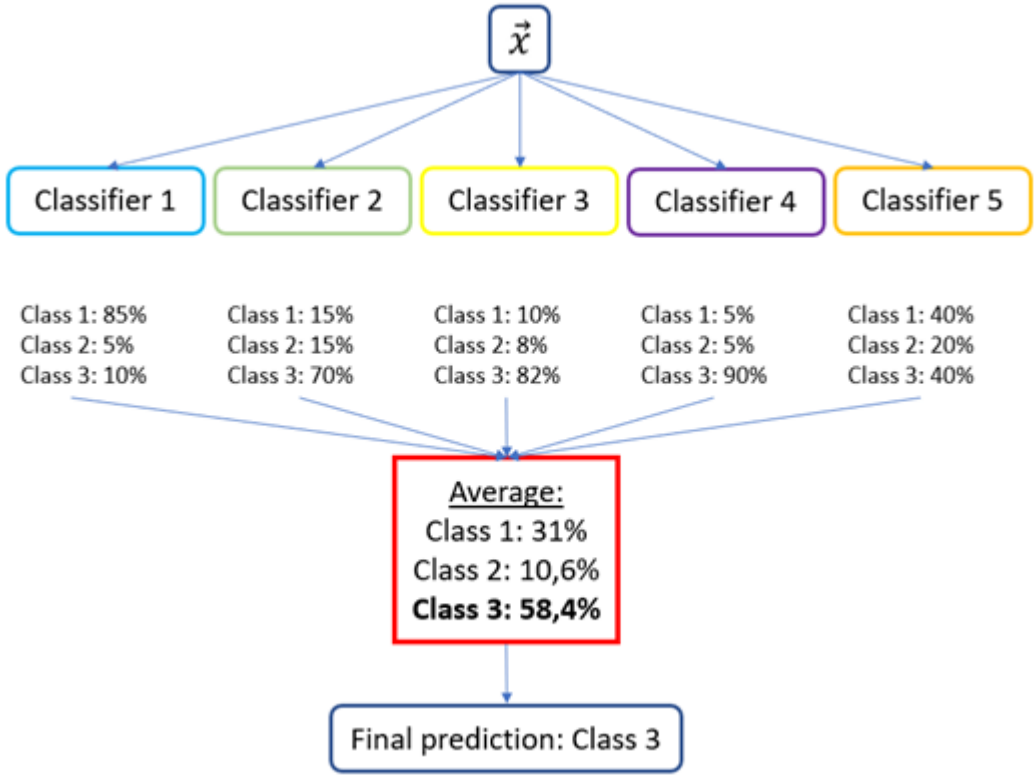

Ilustración 41: Ensemble por media

Se puede observar lo comentado anteriormente, se realiza la suma de las distintas probabilidades obtenidas por los distintos modelos y se calcula la media total de probabilidades, obteniendo así una predicción conjunta. Para realizar esto es muy sencillo simplemente tendríamos que realizar lo siguiente:

```
prob_a = model_resnet50.predict(test_generator)
prob_b = model_efficientnetb7.predict(test_generator)
prob_c = model_efficientnetb5.predict(test_generator)

probabilities = np.add(prob_a, prob_b)
probabilities = np.add(probabilities, prob_c)
```

Ilustración 42: Suma de predicciones

Como se ha comentado al principio, lo que se hace es calcular la predicción de cada modelo y posteriormente realizar la suma de probabilidades.

Para calcular la predicción final del ensemble se calcularía cuál es el índice con máxima probabilidad para cada array.

```
prediction = np.argmax(probabilities,axis=1)
```

Ilustración 43: Predicción Ensemble

Teniendo ya la predicción final comprobaremos si la implementación de este sencillo ensemble resulta en una mejora para nuestro problema de clasificación. Por lo tanto, calcularemos el AUC para nuestro ensemble:

```
from sklearn.metrics import roc_auc_score

roc_auc_score(true_prediction_transformed, prediction_transformed, multi_class='ovr')
```

```
0.9801319581932596
```

Ilustración 44: AUC Ensemble

Podemos observar cómo el resultado AUC vuelve a ser mayor a 0.98 cómo lo era sobre el conjunto de validación. Por lo tanto, un ensemble muy sencillo nos ha ayudado ha paliar dicha perdida de nuestros modelos por separado.

Si mostramos la matriz de confusión resultante de nuestro modelo final, el ensemble, veríamos cómo estaría coloreada de otro color la diagonal de la siguiente manera:

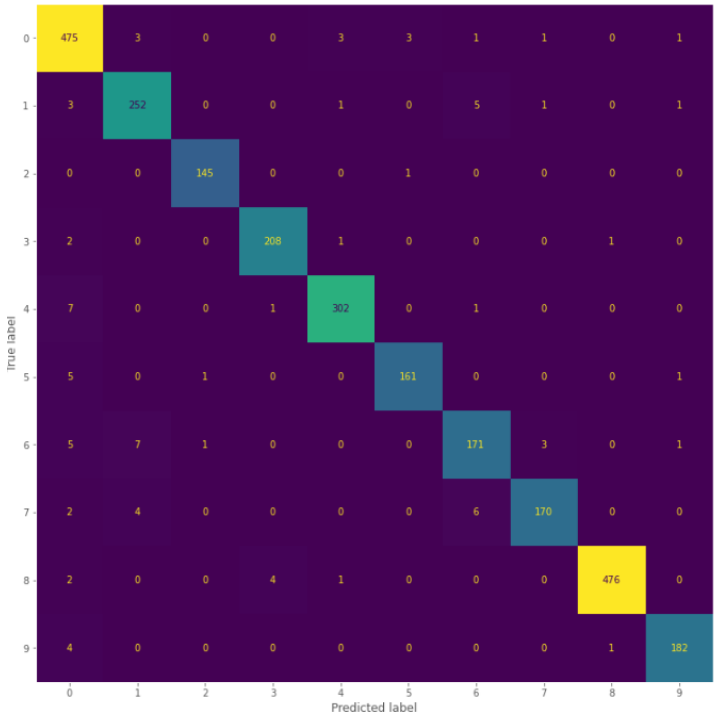

Ilustración 45: Matriz de confusión

Podemos visualizar cómo la mayoría de predicciones se encuentran en la diagonal, la matriz de confusión es muy clarificadora en el sentido de que nos muestra que nuestro modelo final es un muy buen modelo para el problema que se nos había presentado.

7. Conclusión

Para terminar, queremos destacar que la implementación de los diferentes tipos de modelos nos permite sacar una serie de conclusiones:

- Una buena arquitectura en un diseño creado manualmente puede ser muy superior a cualquier modelo preentrenado.
- La problemática que encontramos del punto anterior es que para encontrar dicha arquitectura tienes que realizar muchas pruebas y el entrenamiento se dispara en tiempo de ejecución para encontrar la óptima.
- Los modelos preentrenados solventan muy bien dicha problemática de encontrar la mejor arquitectura, dado que ofrecen un muy buen resultado.
- La agrupación de resultados puede ofrecer un mejor resultado, esto se ve cuando realizamos una agregación muy simple con el ensemble.

En conclusión, estamos muy contentos con el trabajo realizado con este proyecto dado que hemos conseguido entender todos esos puntos y profundizar en Tensorflow/Keras. Además, creemos que el resultado obtenido es muy bueno dado el problema que teníamos al tener muy pocas muestras de ciertas clases. Por lo tanto, la valoración interna que realizamos es muy buena y creemos que la mejor forma de aprender es poniéndonos delante de un problema.