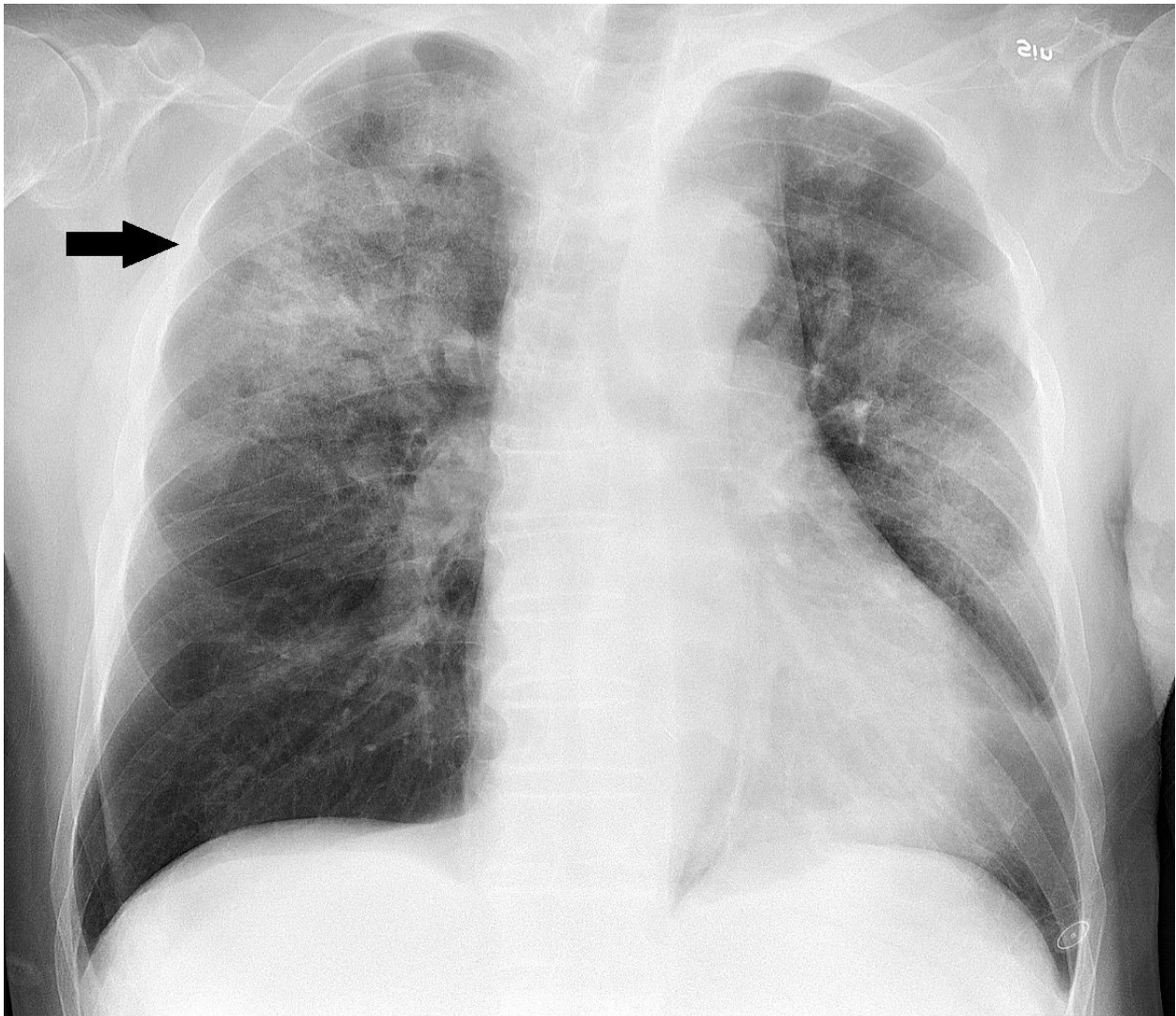


INTELIGENCIA ARTIFICIAL

MODELO DE PREDICCIÓN DE NEUMONÍA

ACTIVIDAD 5



Ramón Íñiguez Bascuas - 21802766

Víctor Hernández Sanz - 21835807

Ruben Ortiz Nieto - 21841860

Índice

Introducción	3
Estado del arte	4
Desarrollo e implementación de la aplicación	5
Entrenamiento del Modelo	9
Resultados obtenidos	13
Conclusiones	16

Introducción

La neumonía es una afección inflamatoria del pulmón que afecta principalmente a los pequeños sacos de aire conocidos como alvéolos. Los síntomas típicamente incluyen una combinación de tos productiva o seca, dolor de pecho, fiebre y dificultad para respirar. El diagnóstico a menudo se basa en los síntomas y el examen físico, donde la radiografía de tórax, los análisis de sangre y el cultivo del esputo pueden ayudar a confirmar el diagnóstico.

Por lo tanto crearemos un clasificador basado en un modelo de aprendizaje automático, CNN (Conventional Neural Network), basado en imágenes introducidas por el médico donde la predicción generada por nuestro clasificador sirva como ayuda para apoyar la decisión tomada por el médico.

Para los siguientes apartados procederemos a explicar los trabajos previos relacionados con el tema, describiremos el proceso de desarrollo realizado junto a sus características, interpretaremos los resultados obtenidos por el modelo entrenado y finalmente las conclusiones que podemos extraer del proyecto

Estado del arte

En este proyecto vamos a desarrollar mediante un CNN (Convolutional Neural Network) con diferentes números de capas y el procesamiento de imágenes, un clasificador de imágenes, que dividirá en Neumonía y No Neumonía un conjunto de radiografías.

Anteriormente se han realizado pruebas sobre similitud entre distintos textos, basándose en las características que presentan cada uno, analizando mediante distintos modelos de clasificación, cuál es la intención del mensaje, no solo analizar en función de las veces que se repiten las palabras, de esta manera se obtienen pruebas con una precisión de más del 95%.

ImageNet consta de más de 14 millones de imágenes que comprenden clases de animales, flores, objetos cotidianos, personas y muchos más. Entrenar un modelo en ImageNet le da la capacidad de adaptarse a la visión a nivel humano, dada la diversidad de datos. “Noisy Student” es un método de aprendizaje semi-supervisado, que logra la mejor precisión con un 88,4% en ImageNet y ganancias sorprendentes en robustez y puntos de referencia. Fue presentado por el equipo de Google en colaboración con la Universidad Carnegie Mellon.

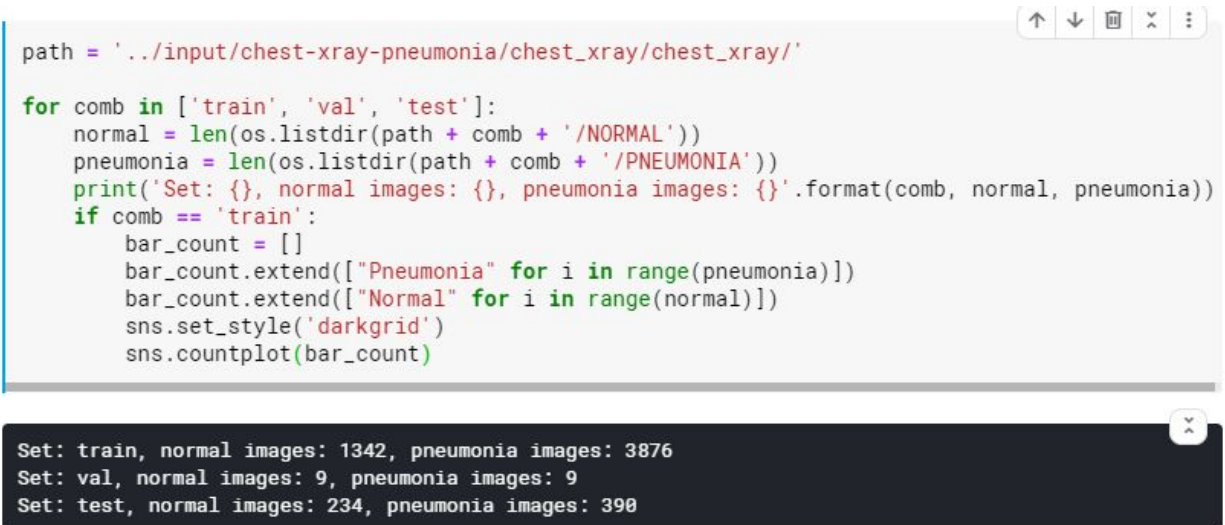
Primero entrenaron un modelo EfficientNet en imágenes etiquetadas de ImageNet y lo usaron como profesor para generar pseudo etiquetas en 300 millones de imágenes sin etiquetar. Luego, los investigadores entrenaron una EfficientNet más grande como modelo de aprendizaje en la combinación de imágenes etiquetadas y pseudo etiquetadas.

Desarrollo e implementación de la aplicación

Para poder desarrollar nuestro clasificador, hemos usado el dataset de 'Kaggle' llamado Chest X-Ray Images (Pneumonia) que dispone de más de 5000 radiografías.

Para esto hemos necesitado emplear el compilador online que proporciona 'Kaggle', que dispone de un modo de alta velocidad gracias al uso de GPUs, lo que reduce muy notablemente el tiempo de cómputo a tan solo unos minutos. Para usar esta característica es necesario estar registrado en 'Kaggle' y seleccionar a la derecha, en 'Settingsfalse' y en 'Accelerator' seleccionar la característica GPU.

Para acceder a los archivos recorreremos todos los directorios y subdirectorios, donde se encuentran todas las imágenes ordenadas según los nombres de 'train', 'test' y 'val', y obtenemos la cantidad de imágenes en cada fichero para poder determinar si el dataset está balanceado o no:

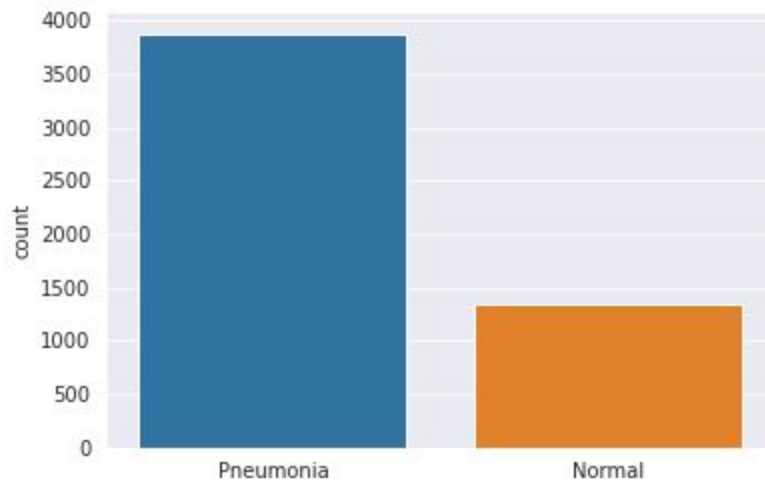


```
path = '../input/chest-xray-pneumonia/chest_xray/chest_xray/'

for comb in ['train', 'val', 'test']:
    normal = len(os.listdir(path + comb + '/NORMAL'))
    pneumonia = len(os.listdir(path + comb + '/PNEUMONIA'))
    print('Set: {}, normal images: {}, pneumonia images: {}'.format(comb, normal, pneumonia))
    if comb == 'train':
        bar_count = []
        bar_count.extend(["Pneumonia" for i in range(pneumonia)])
        bar_count.extend(["Normal" for i in range(normal)])
        sns.set_style('darkgrid')
        sns.countplot(bar_count)
```

Set: train, normal images: 1342, pneumonia images: 3876
Set: val, normal images: 9, pneumonia images: 9
Set: test, normal images: 234, pneumonia images: 390

Donde observamos como en el fichero 'val' y 'test' ambos están balanceados mientras que el fichero 'train' está desbalanceado ya que hay casi el triple de imágenes de una clase que de otra pudiéndose ver en el gráfico posterior:



A continuación procedemos con el preprocesamiento de los datos donde lo primero que haremos es normalizar los datos de sus valores 0 a 255 (pixel) a 0 a 1 ya que el modelo utilizado (Convolutional Neural Network / CNN) converge en menos tiempo de esta forma. Además para evitar el overfitting a la hora de entrenar el modelo realizamos un aumento de los datos de la clase minoritaria, mediante técnicas de reescalado de la imagen y cambio de la orientación para estas imágenes actúan como otra nueva, y añadiendo a esto dividimos las imágenes en 'batch' que nos será útil más tarde a la hora de entrenar el modelo, donde ponemos como estándar del tamaño de la imagen 150x150.

Lo mismo que hemos mencionado anteriormente se aplica a las imágenes de validación con la diferencia de que además añadimos una dimensión más, de forma que nos queda una figura de imagen de (150, 150, 3) lo cual es imperativo para construir posteriormente la red neuronal y se devuelve como un array de numpy.

```

def process_data(batch_size):
    # We normalize the data (cnn converges faster on [0..1] data than [0..255])
    # And in order to avoid overfitting we artificially expand our data set using parameters such
    # as zoom_range, vertical_flip etc.
    train_datagen = ImageDataGenerator(rescale = 1./255, zoom_range = 0.3, vertical_flip = True)
    test_val_datagen = ImageDataGenerator(rescale=1./255)

    # Obtains image with the defined batch size which will be useful later when we fit the model
    train_gen = train_datagen.flow_from_directory(
        directory = path + 'train',
        target_size = (150, 150),
        batch_size = batch_size,
        class_mode = 'binary',
        shuffle = True)

    test_gen = test_val_datagen.flow_from_directory(
        directory = path + 'test',
        target_size = (150, 150),
        batch_size = batch_size,
        class_mode = 'binary',
        shuffle = True)

    # The predictions will be based on one batch size
    test_data = []
    test_labels = []

    for cond in ['/NORMAL/', '/PNEUMONIA/']:
        for img in (os.listdir(path + 'test' + cond)):
            img = plt.imread(path+'test'+cond+img)
            img = cv2.resize(img, (150, 150))
            # Such that the shape of the images is (150, 150, 3)
            img = np.dstack([img, img, img])
            # We normalize the data again ([0..255 --> 0..1])
            img = img.astype('float32') / 255
            if cond=='/NORMAL/':
                label = 0
            elif cond=='/PNEUMONIA/':
                label = 1
            test_data.append(img)
            test_labels.append(label)

    for cond in ['/NORMAL/', '/PNEUMONIA/']:
        for img in (os.listdir(path + 'test' + cond)):
            img = plt.imread(path+'test'+cond+img)
            img = cv2.resize(img, (150, 150))
            # Such that the shape of the images is (150, 150, 3)
            img = np.dstack([img, img, img])
            # We normalize the data again ([0..255 --> 0..1])
            img = img.astype('float32') / 255
            if cond=='/NORMAL/':
                label = 0
            elif cond=='/PNEUMONIA/':
                label = 1
            test_data.append(img)
            test_labels.append(label)

    test_data = np.array(test_data)
    test_labels = np.array(test_labels)

    return train_gen, test_gen, test_data, test_labels

```

Usamos 10 'epochs' y un 'batch_size' de 32:

```
epochs = 10  
batch_size = 32  
  
train_gen, test_gen, test_data, test_labels = process_data(batch_size)
```

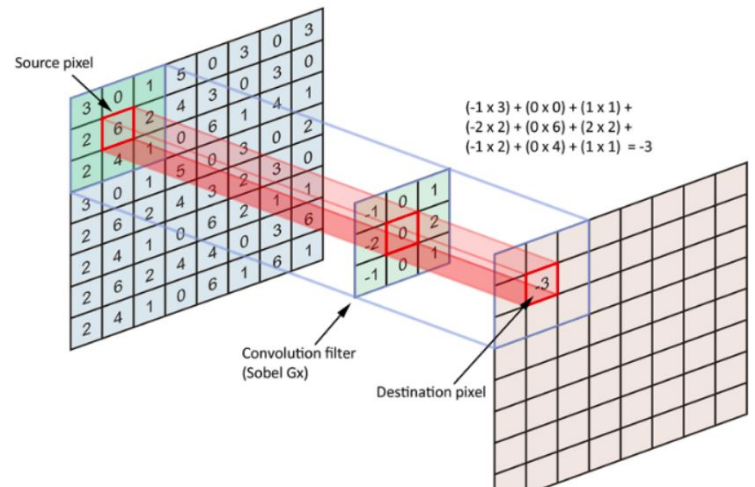
```
Found 5216 images belonging to 2 classes.  
Found 624 images belonging to 2 classes.
```

Entrenamiento del Modelo

Explicación de CNN (Convolutional Neural Network)

-La red toma una imagen de entrada y usa un filtro (o kernel) para crear un mapa de características que describe la imagen.

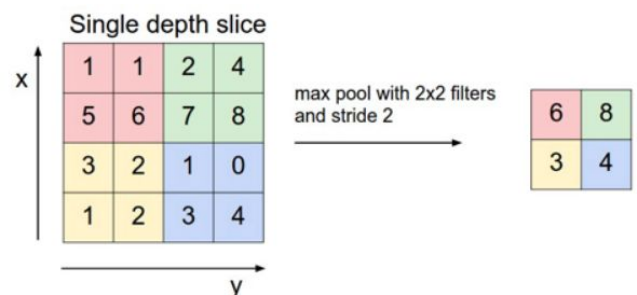
-En la operación de convolución, tomamos un filtro (generalmente matriz de 2x2 o 3x3) y lo deslizamos sobre la matriz de imagen. Los números correspondientes en ambas matrices se multiplican y se suman para obtener un solo número que describe ese espacio de entrada. Este proceso se repite en toda la imagen.



-Usamos diferentes filtros para pasar por alto nuestras entradas y tomar todos los mapas de características, juntarlos como el resultado final de la capa convolucional.

-Luego pasamos la salida de esta capa a través de una función de activación no lineal. El más utilizado es ReLU.

-El siguiente paso de nuestro proceso implica reducir aún más la dimensionalidad de los datos, lo que reducirá la potencia de cálculo requerida para entrenar este modelo. Esto se logra mediante el uso de una capa de agrupación (Pooling Layer). El más utilizado es MaxPooling, que toma el valor máximo en la ventana creada por un filtro. Esto reduce significativamente el tiempo de formación y evita pérdidas de información importante.



Procedemos a diseñar la red neuronal usando un enfoque modular en donde dividiremos las distintas capas de la red en dos bloques básicos, un bloque que actuará como un bloque 'convolutional' que dispondrá de dos capas de neuronas mediante la función 'SeparableConv2D' que a diferencia de la capa 'Conv2D' ya que introduce un número inferior de parámetros y a medida que distintos filtros se aplican a cada capa captura más información. Seguido de la función 'BatchNormalization' que a medida que añadimos más capas a la red neuronal va cobrando más importancia.

Finalmente hacia el final de la red neuronal usamos la función 'dense_block' que está formada por la función 'Dense' y 'Dropout' donde la primera forma la salida de nuestra capa mientras que la última elimina ciertos nodos para reducir la probabilidad de overfitting.

```
def conv_block(filters):
    block = Sequential([
        SeparableConv2D(filters, (3,3), activation='relu', padding='same'),
        SeparableConv2D(filters, (3,3), activation='relu', padding='same'),
        BatchNormalization(),
        MaxPool2D((2,2))
    ])
    return block
```

```
def dense_block(units, dropout_rate):
    block = Sequential([
        Dense(units, activation='relu'),
        Dropout(dropout_rate)
    ])
    return block
```

Como se puede ver en la siguiente imagen construimos una red neuronal formada por cinco 'convolutional block' donde el primer bloque usa dos capas de la función 'Conv2D' ya que nos interesa que capture detalles generales como puede ser esquinas, formas, etc. Y la última capa está formada por una única neurona, la cual nos indica la clase a la que pertenece la imagen con una activación distinta, donde estas capas ocultas forman parte del 'fine tuning' del modelo junto a:

- Como función de activación de las neuronas usamos 'relu' en vez de otros algoritmos ya que este funciona mejor.

- Como optimizador usamos 'Adam' en vez de otros como 'SGD' ya que 'Adam' tiene un coste de entrenamiento menor lo que permite converger antes a pesar de tardar un poco más.

```
model = Sequential([
    # First (input) convutional block
    Input(shape=(150,150, 3)),
    Conv2D(16, (3,3), activation='relu', padding='same'),
    Conv2D(16, (3,3), activation='relu', padding='same'),
    MaxPool2D((2,2), padding = 'same'),

    # Second convutional block
    conv_block(32),

    # Third convutional block
    conv_block(64),

    # Fourth convutional block
    conv_block(128),
    Dropout(0.2),

    # Fifth convutional block
    conv_block(256),
    Dropout(0.2),

    Flatten(),
    dense_block(512, 0.7),
    dense_block(128, 0.5),
    dense_block(64, 0.3),
    Dense(1, activation = 'sigmoid')
])

model.summary()
model.compile(loss='binary_crossentropy', optimizer= 'adam', metrics=['accuracy'])

# Callbacks

checkpoint = ModelCheckpoint(filepath='best_weights.h5', save_best_only=True)
lr_reduce = ReduceLROnPlateau(monitor='val_loss', factor=0.3, patience=2, verbose=2, mode='max')
early_stop = EarlyStopping(patience=10, restore_best_weights=True)
```

Además para optimizar aún más el modelo usamos los siguientes checkpoints:

- ModelChekpoint: Al entrenar se requieren muchos epochs para conseguir un buen resultado, por lo tanto, es mejor hacer una copia del mejor rendimiento del modelo.
- ReduceLROnPlateau: Nos interesa reducir el ratio de aprendizaje (learning rate) para que converja de la mejor manera.
- EarlyStopping: Durante el entrenamiento la diferencia entre el error de entrenamiento y validación empieza a incrementar en vez de decrementar, (síntoma de overfitting) por lo que conviene parar el aprendizaje.

El resultado de los parámetros al crear el modelo es el siguiente:

```
Model: "sequential_85"
-----
Layer (type)                Output Shape              Param #
-----
conv2d_24 (Conv2D)          (None, 150, 150, 16)     448
conv2d_25 (Conv2D)          (None, 150, 150, 16)     2320
max_pooling2d_51 (MaxPooling (None, 75, 75, 16)      0
-----
sequential_78 (Sequential)   (None, 37, 37, 32)       2160
sequential_79 (Sequential)   (None, 18, 18, 64)       7392
sequential_80 (Sequential)   (None, 9, 9, 128)        27072
dropout_42 (Dropout)         (None, 9, 9, 128)        0
sequential_81 (Sequential)   (None, 4, 4, 256)        103296
dropout_43 (Dropout)         (None, 4, 4, 256)        0
flatten_12 (Flatten)         (None, 4096)             0
sequential_82 (Sequential)   (None, 512)              2097664
sequential_83 (Sequential)   (None, 128)              65664
sequential_84 (Sequential)   (None, 64)               8256
dense_42 (Dense)             (None, 1)                65
-----
Total params: 2,314,337
Trainable params: 2,313,377
Non-trainable params: 960
-----
```

Resultados obtenidos

Entrenamos el modelo:

```
hist = model.fit(
    train_dgen, steps_per_epoch = train_dgen.samples // batch_size,
    epochs = epochs,
    validation_data = test_dgen,
    validation_steps = test_dgen.samples // batch_size,
    callbacks=[checkpoint, early_stop, lr_reduce])
```

```
Epoch 1/10
163/163 [=====] - 78s 476ms/step - loss: 0.4865 - accuracy: 0.8081 - val_loss: 0.6787 - val_accuracy: 0.6234
Epoch 2/10
163/163 [=====] - 77s 471ms/step - loss: 0.2836 - accuracy: 0.8844 - val_loss: 0.6748 - val_accuracy: 0.6201
Epoch 3/10
163/163 [=====] - 77s 474ms/step - loss: 0.2554 - accuracy: 0.8990 - val_loss: 0.6482 - val_accuracy: 0.6283

Epoch 00003: ReduceLROnPlateau reducing learning rate to 0.0003000000142492354.
Epoch 4/10
163/163 [=====] - 77s 469ms/step - loss: 0.2187 - accuracy: 0.9181 - val_loss: 0.5163 - val_accuracy: 0.7023
Epoch 5/10
163/163 [=====] - 77s 472ms/step - loss: 0.2070 - accuracy: 0.9266 - val_loss: 0.2954 - val_accuracy: 0.8914

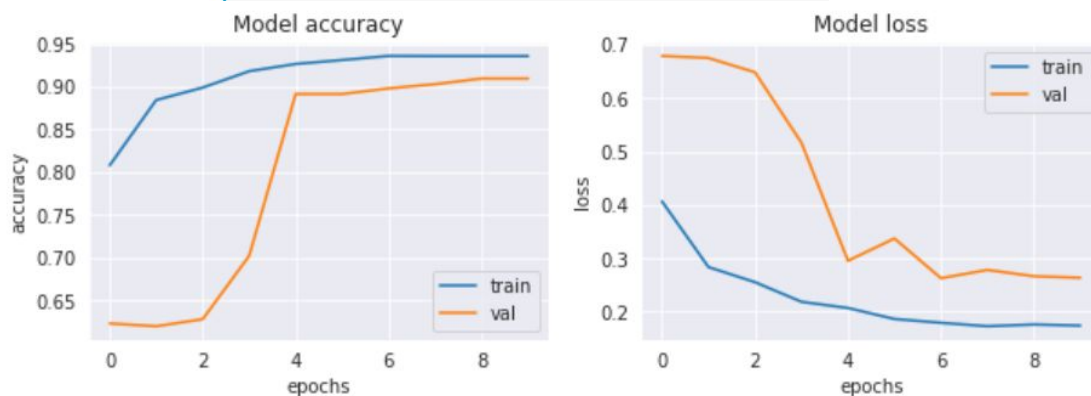
Epoch 00005: ReduceLROnPlateau reducing learning rate to 9.000000427477062e-05.
Epoch 6/10
163/163 [=====] - 77s 471ms/step - loss: 0.1865 - accuracy: 0.9312 - val_loss: 0.3372 - val_accuracy: 0.8914
Epoch 7/10
163/163 [=====] - 77s 470ms/step - loss: 0.1794 - accuracy: 0.9360 - val_loss: 0.2629 - val_accuracy: 0.8980

Epoch 00007: ReduceLROnPlateau reducing learning rate to 2.700000040931627e-05.
Epoch 8/10
163/163 [=====] - 77s 473ms/step - loss: 0.1729 - accuracy: 0.9358 - val_loss: 0.2783 - val_accuracy: 0.9030
Epoch 9/10
163/163 [=====] - 77s 475ms/step - loss: 0.1763 - accuracy: 0.9358 - val_loss: 0.2665 - val_accuracy: 0.9095

Epoch 00009: ReduceLROnPlateau reducing learning rate to 8.100000013655517e-06.
Epoch 10/10
163/163 [=====] - 78s 479ms/step - loss: 0.1741 - accuracy: 0.9358 - val_loss: 0.2637 - val_accuracy: 0.9095
```

```
fig, ax = plt.subplots(1, 2, figsize=(10, 3))
ax = ax.ravel()

for i, met in enumerate(['accuracy', 'loss']):
    ax[i].plot(hist.history[met])
    ax[i].plot(hist.history['val_' + met])
    ax[i].set_title('Model {}'.format(met))
    ax[i].set_xlabel('epochs')
    ax[i].set_ylabel(met)
    ax[i].legend(['train', 'val'])
```



Como se puede apreciar en la primera gráfica, observamos es que a partir de un número de epoch de 8 el modelo ya converge y vemos que no estamos ante un caso de 'overfitting' o 'underfitting' ya que las curvas corresponden con un modelo bien entrenado y la distancia entre ambas curvas decrementa a medida que avanzamos. Además en la segunda gráfica también converge y vemos como la pérdida del modelo sigue el mismo estilo de curva que la primera de forma inversa.

```
from sklearn.metrics import accuracy_score, confusion_matrix

preds = model.predict(test_data)

acc = accuracy_score(test_labels, np.round(preds))*100
cm = confusion_matrix(test_labels, np.round(preds))
tn, fp, fn, tp = cm.ravel()

print('CONFUSION MATRIX -----')
print(cm)

print('\nTEST METRICS -----')
precision = tp/(tp+fp)*100
recall = tp/(tp+fn)*100
print('Accuracy: {}'.format(acc))
print('Precision: {}'.format(precision))
print('Recall: {}'.format(recall))
print('F1-score: {}'.format(2*precision*recall/(precision+recall)))

print('\nTRAIN METRIC -----')
print('Train acc: {}'.format(np.round((hist.history['accuracy'][-1])*100, 2)))
```

```
CONFUSION MATRIX -----
[[188  46]
 [ 11 379]]

TEST METRICS -----
Accuracy: 90.86538461538461%
Precision: 89.17647058823529%
Recall: 97.17948717948718%
F1-score: 93.00613496932515

TRAIN METRIC -----
Train acc: 93.58
```

Finalmente podemos observar la matriz de confusión como la gran mayoría de casos están bien predichos donde tenemos un acierto del 90,86% en el test mientras que en el

entrenamiento tenemos un acierto del 93,58% lo cual indica que podríamos obtener ligeramente mejores hiperparametros para que el porcentaje de acierto fuera mejor que el de entrenamiento, pero, ya que la diferencia es muy leve, podemos asegurar que no estamos ante un caso ni de 'overfitting' ni de 'underfitting'.

Conclusiones

En base a lo que hemos podido observar tras la realización de este proyecto, nos hemos dado cuenta que es muy importante partir de un dataset balanceado.

Tras realizar el proceso de diseño y creación de la red neuronal que hemos empleado para nuestro clasificador, hemos podido aprender cómo se deben de tratar los datos correctamente y diversas técnicas para balancear el set de datos, como puede ser rotar levemente las imágenes, ampliarlas o alejarlas, para que el filtrado de las características varíe y deba entrenar nuevamente y no se produzca 'Overfitting'.

Además podemos ver como el modelo empleado para las el aprendizaje es muy fiable ya que alcanza una 'accuracy' de 90%, y si de verdad quisiéramos obtener la mejor red neuronal posible, tendríamos que experimentar con una gran cantidad de capas ocultas además de probar con una gran cantidad de hiperparámetros, lo cual sería una labor muy cara tanto en tiempo real como en tiempo de cómputo.