ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE GRANADA

Visión por Computador

— Proyecto Final: CoronaHack-Chest X-Ray-Dataset —

Santiago González Silot
Ahmed Brek Prieto

Diciembre, 2021

Curso 2021-22

sgonzalezsilot@correo.ugr.es
ahmed2bp@correo.ugr.es

Índice

Índice de figuras

Índice de cuadros

1. Definición del problema

Desde finales del año 2019, la población de todo el planeta ha sufrido las graves consecuencias de la pandemia mundial causada por el COVID-19 que, de hecho, aún sigue vigente. Al tratarse de una enfermedad que ha afectado tan directamente a todos los países, ha surgido un gran interés en tratar de entender mejor este virus y los síntomas que provoca en el organismo humano. Desde prácticamente el inicio de su expansión, se supo que afecta más gravemente a aquellas personas que sufren problemas respiratorios, independientemente de la edad y otros factores determinantes. De hecho, este virus puede causar enfermedades graves como la neumonía [1], sobre la que centraremos nuestra atención. Dicho esto, se puso especial atención en la evolución de la enfermedad en este grupo de personas. Asimismo, comenzó a ser de suma importancia detectar si los pacientes estaban sanos o habían desarrollado una neumonía. Para su detección, lo más común es realizar radiografías del sistema respiratorio. Precisamente de esto trata el dataset elegido para realizar este proyecto. Este contiene 5.933 imágenes resultantes de radiografías de pacientes, algunos que sufren neumonía y otros que permanecen sanos. El reto que propone este conjunto de datos consiste en determinar si un paciente sufre una neumonía o no, a partir de una radiografía. Se trata de una pregunta muy relevante a la hora de estudiar la gravedad de una persona y el tipo de atención médica que debe recibir, por lo que resulta ser un reto muy interesante y útil en la práctica. Nos encontramos entonces frente a un problema de clasificación binaria.

1.1. Descripción del conjunto de datos

El conjunto de datos se ha obtenido de esta web. Las imágenes se distribuyen en dos carpetas ya separadas (una para el *train* y otra para el *test*). La primera, contiene 5.309 ejemplos y la segunda, 624, sumando un total de 5.933 casos. Además, para indicar las etiquetas de la muestra, encontramos un fichero en formato *csv* que contiene el nombre de cada imagen, su etiqueta y cierta información adicional (si la neumonía es causada por un virus o una bacteria, ente otras). Este fichero nos permite comprobar que existen 1.576 casos de pacientes sanos frente a 4.334 casos de pacientes que sufren neumonía. Observamos así que existe cierto desbalanceo en los datos, pues existen muchos menos datos de la clase de pacientes sanos (26'67 %). Este problema se tratará más adelante.

1.2. Aclaración

Se ha de mencionar que, con el objetivo de no extender excesivamente la memoria del proyecto, tan solo se van a explicar los conceptos más relevantes en la resolución de este problema en concreto y no conceptos generales ya estudiados en la parte teórica de la asignatura.

2. Procedimiento

Para la creación del modelo predictivo, se debe seguir un proceso que consta de tres partes. Primero, se deben leer los datos y procesarlos de tal manera que puedan ser utilizados por el modelo. Además, se ha de aplicar el preprocesado necesario y tratar los problemas que estos puedan presentar como, por ejemplo, el desbalanceo existente en las clases ya observado.

En segundo lugar se aplicará *Transfer Learning* con *Resnet50* entrenada con *Imagenet* como modelo básico (es decir, se adaptará la salida de esta red a nuestro problema, para ello se añadirá una última capa junto a una función de activación sigmoide) y se estimará su bondad según el conjunto de validación. Más adelante, se aplicarán mejoras (añadiendo capas a la red o realizando modificaciones sobre los datos de entrada) para crear una red más compleja y obtener mejores resultados.

Finalmente se estudiarán los resultados que ofrece el modelo definitivo, del cual se estimará el error de generalización.

2.1. Lectura de datos

Como se ha explicado, el primer paso consiste en leer los datos del *dataset*. Ya se ha mencionado que los datos vienen separados en una carpeta de entrenamiento y otra de prueba. Tal y como se ha explicado alguna vez en teoría, suele ser preferible mezclar todos los datos y realizar una nueva partición aleatoria, con el fin de evitar posibles sesgos existentes en la partición dada. Para ello, se han mezclado todas las imágenes en una misma carpeta llamada "unclassified", que será leída directamente por el programa.

Obviando ese detalle, el ejecutable se encarga de realizar la lectura de datos por completo. Primero, se lee el archivo en formato *csv*, donde se indica el nombre de cada imagen y su etiqueta, y se almacena en un *dataframe*. De ese *dataframe* se separa un conjunto de train y otro de test (reservando un 20 % de los datos para el conjunto de prueba), utilizando la función *train_test_split()* de *Scikit-Learn*. Mediante el parámetro *stratify* se indica que se desea mantener la proporción en los conjuntos según su etiqueta, para garantizar que el conjunto de *test* tenga cierta cantidad de datos de la clase minoritaria.

A continuación, se crean las carpetas de entrenamiento y prueba (en caso de que ya existan, se borran primero). Se copian las imágenes en la carpeta correspondiente según el conjunto al que pertenezca cada una. Así, se tienen ambos conjuntos de datos separados en carpetas.

Hecho esto, queda un último paso. En un primer análisis del *dataset*, se ha encontrado un notable desbalanceo en los datos. Para tratarlo, existen distintas alternativas, como pueden ser *upsampling* (generar nuevos datos de la clase minoritaria a partir de los ya existentes), *downsampling* (eliminar una parte de los datos de la clase predominante para encontrar un mayor equilibrio) o el uso de métricas de error adaptadas a estos casos (como por ejemplo, *balanced accuracy*). Para este proyecto, se ha optado por realizar *upsampling*, para no tener que renunciar a parte de la muestra, y por resultar una opción más atractiva que simplemente utilizar alguna métrica concreta (esto sería tratar el error de forma tangencial, pero no constituye una solución como tal). Otra razón es que la generación de nuevos datos en problemas de visión por computador es una práctica bastante típica (*data augmentation*), por lo que no resulta demasiado "arriesgado" crear nuevas instancias, siempre y cuando se haga con cierta precaución.

El momento para aplicar dicho *upsampling* es durante la lectura de los datos. Para ello, se utilizará un objeto de la clase *ImageDataGenerator* que, como su nombre indica, permite generar nuevas instancias a partir de un conjunto de datos. Los parámetros que recibe el constructor determinan las transformaciones que se aplicarán a cada imagen para generar las nuevas. En este caso, se opta por los siguientes:

- *rotation_range*: indica el rango de la rotación máxima permitida. Toma valor 0'2 por tratarse de un valor mediano, sin ser demasiado grande como para volcar demasiado la radiografía. Recordemos que se trata de un umbral máximo, por lo que, en general, las rotaciones serán menores que dicho valor.
- *width_shift_range*: determina el rango de desplazamiento horizontal máximo. En este caso, se toma un valor de 0'025, bastante pequeño, ya que las imágenes del *dataset* no permiten demasiado desplazamiento en el eje X (la radiografía se saldría de la imagen si el desplazamiento es demasiado grande).
- *zoom_range*: indica el máximo zoom permitido aplicable sobre las imágenes. Toma un valor de 0'1 por razones análogas al parámetro anterior.

Existen otras transformaciones útiles que se suelen aplicar. Se explica a continuación por qué no han sido utilizadas:

- **Vertical Flip**: Esta transformación se descartó en cuanto se supo la índole del *dataset*, ya que al tratarse de radiografías médicas, si se voltean verticalmente se van a introducir imágenes "inútiles" que solo van a empeorar el modelo, pues este nunca va a tener que predecir radiografías volteadas.

- **Horizontal Flip**: Si bien este parámetro podría llegar a tener sentido, ya que las mitades izquierda y derecha de la radiografía son similares, se ha comprobado que es mala idea ya que al poder observarse en la radiografía el corazón (el cual se encuentra principalmente en el lado izquierdo del pecho), si se voltea la imágen cambia de lugar lo cual puede causar un mal funcionamiento del modelo. Se muestra un ejemplo a continuación:

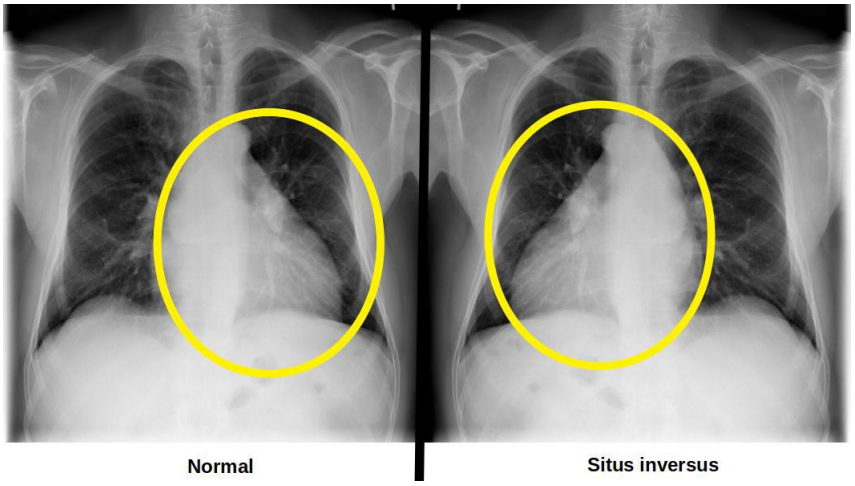


Figura 1: Horizontal Flip en radiografías

- **Shear Range**: Este parámetro es parecido a *rotation_range*. Sin embargo, a diferencia de este, *shear_range* "estira" las imágenes, por lo que las deforman aún más y no parece la mejor opción.
- **Desplazamiento vertical**: Con *height_shift_range* se desplaza verticalmente la imagen, lo cual en este conjunto de datos no es interesante ya que podría eliminar zonas importantes de la imagen que aportan información útil.

Una vez creado este objeto, se leen las imágenes del conjunto de entrenamiento. Después, se aplica el upsampling sobre las imágenes leídas pertenecientes a la clase minoritaria (la de radiografías de pacientes sanos), añadiendo sus etiquetas al vector correspondiente. Se mezclan los vectores de imágenes originales y generadas y se obtiene el conjunto de entrenamiento final, sin desbalanceo de clases. Cabe añadir que, por cada imagen original, se crean una adicional, obteniendo así un conjunto final de entrenamiento con 3783 radiografías de pacientes sanos y 3467 de pacientes con neumonía [1].

2.2. Modelo base

Para el modelo base se utilizará ResNet50 entrenada con la base de datos Imagenet. Se trata de una base de datos compuesta por más de 14 millones de imágenes de distintas índoles. Es uno de los *datasets* más utilizados en el ámbito del *deep learning* tanto para clasificación como para detección de objetos.

Por otro lado ResNet50 es una de las redes más populares y ampliamente utilizadas, que se caracteriza por el uso de bloques residuales y *skip*. A continuación se puede observar la arquitectura de ResNet50:

[1] Las imágenes generadas sólo afectan al conjunto de entrenamiento, sin incluirse ninguna en el conjunto de *test*, pues esto añadiría ruido en este conjunto y supone una mala praxis.

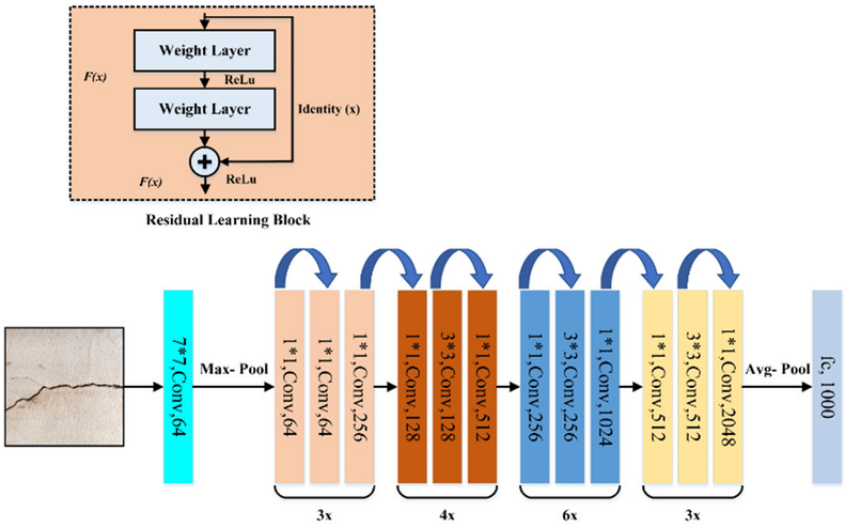


Figura 2: Arquitectura de Resnet50

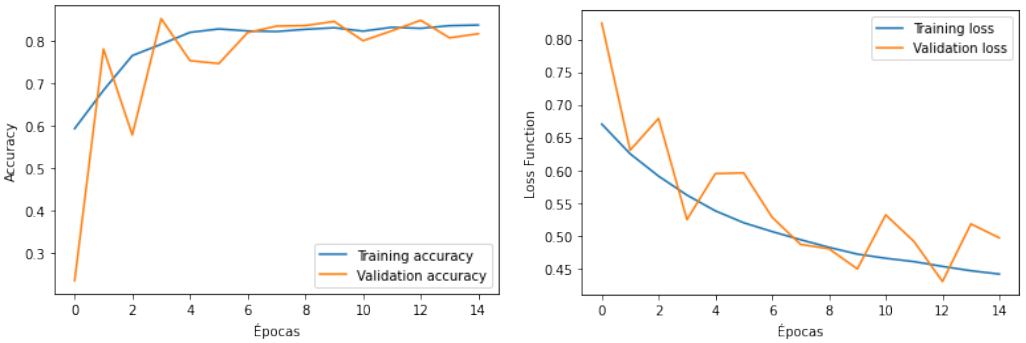
Para el modelo base se ha adaptado ResNet50 a este problema. Es decir, se ha añadido una capa final tras las correspondientes al modelo ResNet50 (las cuales no se entrenan). Esta capa tiene una única neurona (que determina la salida del problema) junto a una capa extra de activación de tipo sigmoide, por tratarse de un problema de clasificación binaria [2][3]. Tras esto, el número de parámetros de la red es de 23,589,761, de los que 2,049 son entrenables. Además cabe mencionar que las imágenes de entrada son de tamaño (224,224,3).

Otro aspecto relevante es la función de pérdida utilizada para optimizar los pesos de la red. En este caso, se ha hecho uso de la entropía cruzada binaria (*binary cross entropy*, o *log loss*). Se ha tomado esta función por tratarse de un problema de clasificación binaria y ser la medida típica utilizada frente a estos casos [4].

Una vez definida la función de pérdida, queda indicar el algoritmo de optimización a usar. Para este modelo, se ha optado por usar el algoritmo Adam. En pocas palabras, consiste en una extensión del ya conocido gradiente descendente estocástico (SGD), enfocado a aproximar los pesos de redes neuronales [5]. Se trata de una buena opción ya que en el propio *paper* original se demuestra de forma empírica que ofrece buenos resultados sin necesidad de largos tiempos de ejecución [6].

Se ha entrenado el modelo durante 15 épocas para poder obtener los resultados iniciales y se han obtenido los siguientes resultados.

Modelo	Validation Loss	Validation Accuracy
Modelo Base	0'5011	0'8288



(a) Accuracy (b) Loss

Figura 3: Evolución del modelo base

Se observa como el modelo efectivamente aprende a lo largo de las épocas, pues el *accuracy* aumenta y disminuye el valor de la función de pérdida. Por otro lado, cabe destacar que con la red preentrenada ya se obtiene una precisión del 80 %, relativamente aceptable, en el conjunto de validación. Además, cabe añadir que se aprecian ciertas fluctuaciones en la evolución del valor de la función de pérdida, haciendo que esta sea algo irregular.

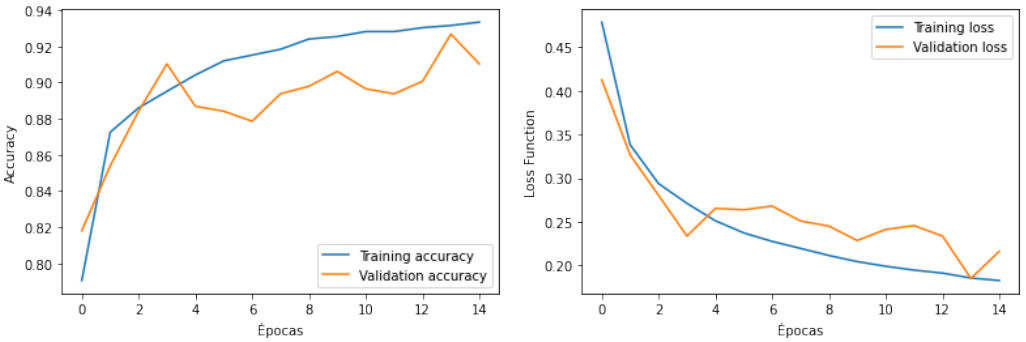
2.3. Mejoras

2.3.1. Normalización de la entrada

En primer lugar, para mejorar el modelo base se va a aplicar normalización de datos. Para ello, se utilizará la clase *ImageDataGenerator* que proporciona *Keras*. Esta modificación consiste en normalizar las instancias del conjunto de entrenamiento utilizado (hacer que tengan media 0 y varianza unitaria). Además, los datos del conjunto de *test* deben ser normalizados con los mismos parámetros que los datos de entrenamiento. Esto se debe a que, a priori, no conocemos los datos de prueba ni su distribución.

La normalización es de gran utilidad en el ámbito de Machine Learning para ciertos modelos. El motivo es que distintos datos de entrada tienen distintos rangos de valores. Así, esa diferencia de rangos puede hacer que cada dato influya más o menos de lo que debería en el proceso de optimización, lo cual se puede evitar aplicando normalización. Por ejemplo, en un modelo clásico de Machine Learning en el que se tenga como entrada la edad de una persona y su salario, es fácil comprobar que el salario será siempre un número mucho más grande que la edad, por lo que influirá más en el proceso de optimización. En este caso, al tratarse de imágenes, existen diferentes intensidades de brillo, por lo que se intuye que normalizarlas puede ser una buena práctica.

Modelo	Validation Loss	Validation Accuracy
Modelo Base	0’2127	0’9061



(a) Accuracy (b) Loss

Figura 4: Evolución del modelo tras aplicar normalización de la entrada

Tras aplicar normalización a los datos de entrada, el rendimiento del modelo aumenta bastante, llegando a alcanzar un *accuracy* del 90 %. También se observa que la función de pérdida disminuye de forma más regular (las fluctuaciones mencionadas anteriormente parecen haberse suavizado). Así, el aprendizaje parece ser en cierto modo más robusto.

2.3.2. Aumento de datos de entrada

Más conocida como *data augmentation*, se trata de otra técnica que influye únicamente en los datos de entrada. Consiste en aumentar el conjunto de datos utilizado para entrenar el modelo generando nuevas instancias. Estas instancias se crean aplicando transformaciones a las del conjunto original, tal y como se hizo para tratar el desbalanceo. Las más comunes consisten en transformaciones geométricas, giros, modificación de color, recortes, rotaciones e introducción de ruido[3]. El principal objetivo de esta técnica es incrementar la robustez del modelo, haciendo que éste sea más genérico sin necesidad de recopilar nuevos datos.

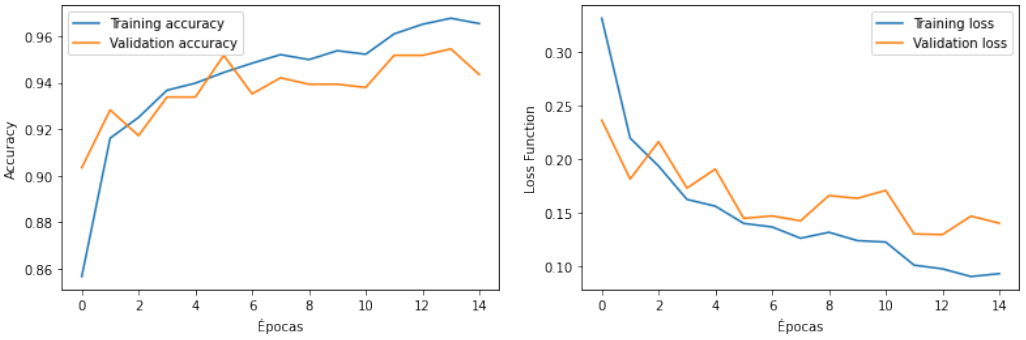
A pesar de que se trata de una práctica bastante recomendable, no será aplicada en este proyecto. La razón es simple: ya se ha realizado cierto aumento de datos sobre una clase concreta para balancearlas, por lo que generar nuevas instancias a partir de algunos datos que no son originales podría introducir demasiado ruido en la muestra. Además, el *dataset* no es precisamente pequeño, por lo que no existe una necesidad real de generar nuevos datos.

2.3.3. Inclusión de capas adicionales

Un aspecto a considerar es que la utilización de una red preentrenada no implica que esta se adapte al *dataset* que se desea resolver. Así, sus pesos no se han entrenado en función de los datos utilizados. Para hacer que la red aprenda en base a las imágenes utilizadas, se necesitan añadir capas posteriores que no hayan sido entrenadas aún, cuyos pesos se calculen basándose en los datos concretos. Normalmente, las capas que se suelen añadir son capas convolucionales y densas. Para este caso, la inclusión de capas convolucionales no resulta una idea demasiado atractiva, pues la salida del modelo base utilizado ya tiene una dimensionalidad baja (y este tipo de capas la reduce aún más). Se podría realizar aumento de dimensionalidad, aunque al tratarse de una medida algo más "arriesgada" y teniendo en cuenta que se está alcanzando una precisión relativamente aceptable, se ha optado por agregar capas densas al modelo. Se trata de capas que conectan cada neurona de un tensor con todas las neuronas del siguiente y, cuyos pesos, como siempre, deben ajustarse.

Concretamente, se han añadido tres capas densas: una de tamaño 100, otra de 50 para ir reduciendo la dimensionalidad, y una última capa que se adapta a la salida del problema (su salida es de una neurona, pues se trata de un problema de clasificación binaria). Los resultados obtenidos son:

Modelo	Validation Loss	Validation Accuracy
Modelo Base	0'1492	0'9429



(a) Accuracy (b) Loss

Figura 5: Evolución del modelo tras el aumento de la profundidad

Encontramos una mejora de un 4 % en la precisión dentro del conjunto de validación y una disminución de 0'0635 en el valor de la función de pérdida, por lo que supone una buena mejora. Además, la distancia entre las curvas de entrenamiento y validación se reduce, y su forma se asemeja algo más, por lo que parece reducirse el sobreajuste en cierta medida.

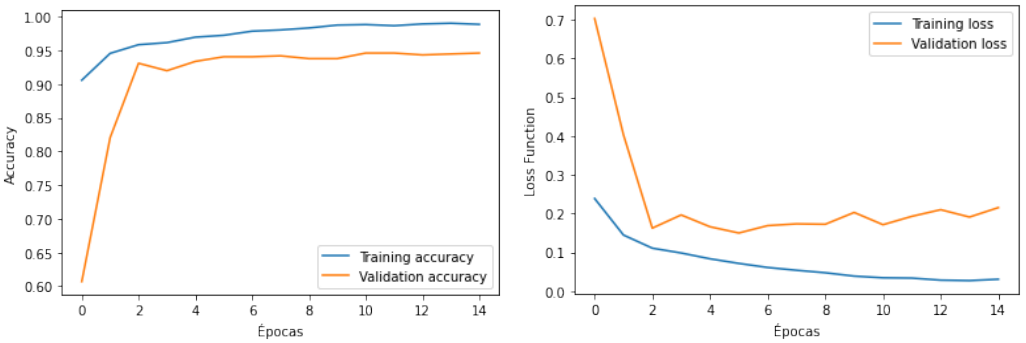
2.3.4. Batch Normalization y Dropout

En los problemas de aprendizaje automático, un aspecto muy importante es la regularización a la hora de ajustar modelos. En el campo de las redes neuronales convolucionales, una opción para aplicar regularización a la red es añadir capas de *batch-normalization*. A continuación, se van a añadir capas de Batch Normalization para ayudar a minimizar en la mayor medida posible el sobreajuste en el aprendizaje. Para la inclusión de capas de este tipo se ha utilizado la clase *BatchNormalization*.

El *Dropout* tiene el mismo objetivo que BN, es decir, lograr una mayor regularización de la red y evitar el sobreajuste. Las capas de este tipo seleccionan aleatoriamente un porcentaje de neuronas activas, en cada *minibatch*, dejando el resto "inactivas". Así, se fuerza de cierta manera a las neuronas a que se especialicen en un tipo de información concreta e independiente del resto. Esto se debe a que, como las redes neuronales suelen tener muchas neuronas, irán aprendiendo de forma "conjunta", influenciándose unas a otras. Entonces, si una neurona en concreto no realiza un aprendizaje correcto, pero el resto sí, esta carencia se verá compensada y no se intentará solventar pues, en general, las neuronas están aprendiendo bien. Desactivando ciertas neuronas se aumenta la independencia de las que quedan activas respecto del resto, y por tanto, tienden a aprender información más concreta.

Más concretamente, se añaden una capa de *batch normalization* tras las salida del modelo básico y dos capas de *dropout*, una después de cada capa densa. Esta decisión se debe a que suele ser adecuado incluir este tipo de capas depués de aquellas que realmente "aprenden" (es decir, capas densas y convolucionales).

Modelo	Validation Loss	Validation Accuracy
Modelo Base	0'2127	0'9388



(a) Accuracy (b) Loss

Figura 6: Evolución del modelo tras aplicar Dropout y Batch Normalization

Se pueden observar dos aspectos tras aplicar esta mejora. Por un lado, los valores de precisión y función de pérdida no parecen variar demasiado. Sin embargo, sí que encontramos cierta diferencia en la morfología de las curvas. A pesar de que las escalas son distintas (la gráfica de la mejora anterior está más "ampliada"), sí que la evolución de la función de pérdida parece presentar fluctuaciones algo menores. Esto sugiere que el aprendizaje es algo más robusto, por lo que sí que se ha logrado aplicar regularización en cierta medida.

2.3.5. Early Stopping

Por último, y no por ello menos importante, es conveniente probar *early stopping*. Una vez se ha confeccionado el modelo, se puede observar que, en las curvas, no siempre se detiene el entrenamiento en el máximo hallado, sino que, a veces, continúa un poco la ejecución y empeoran los resultados finales. La técnica de *early stopping* surge para solventar esta problemática. Consiste en "almacenar" los pesos donde se obtuvo el mejor *accuracy* de las últimas iteraciones y devolver este modelo, pudiendo recuperar su mejor versión antes de que empezase a empeorar con las últimas iteraciones.

En este caso, parece una medida muy útil, ya que en varias gráficas se observa cómo, dadas ciertas fluctuaciones en las últimas épocas, el modelo puede disminuir su *accuracy* en el conjunto de validación al final del entrenamiento, alejándose de un máximo recientemente encontrado. Aplicando *early stopping* se podría tratar este problema.

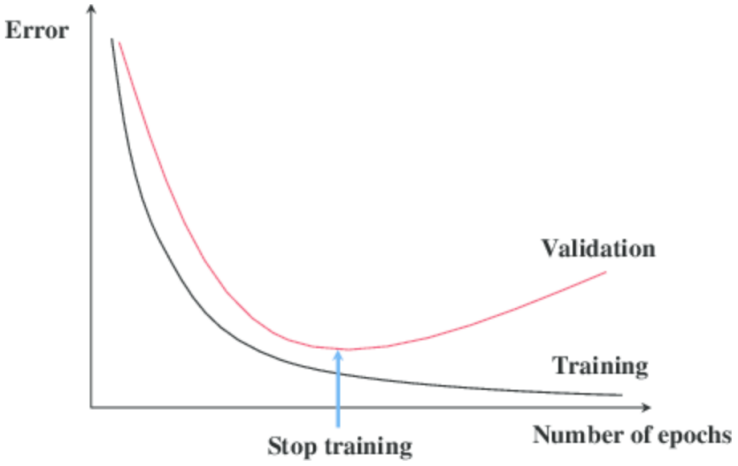


Figura 7: *Early Stopping*

La clase *EarlyStopping* que ofrece *Keras* aporta multitud de parámetros para configurar la

política de early stopping. A continuación muestro los parámetros usados:

- *monitor* = 'val_loss'. Este parámetro determina el valor a valora ser monitorizado. El motivo de utilizar el error de validación en lugar de la precisión de validación es que el error de validación cuantifica la certeza del modelo sobre una predicción (valor cercano a 1 en la clase correcta y cercano a 0 en el resto de clases). En cambio la precisión solo tiene en cuenta el número de precisiones correctas. Por eso es peor utilizar una métrica que use predicciones estrictas. En cualquier caso siempre se ha de tener en cuenta el conjunto de validación y no de entrenamiento si no no tendría sentido el *Early Stopping*.
- *min_delta*=1e-3. Cambio mínimo del valor monitorizado que se considera como una mejora.
- *patience* = 8. Número de epocas sin mejora antes de que se finalice el entrenamiento.
- *restore_best_weights* = True. Para recuperar los mejores pesos encontrados durante el entrenamiento.

3. Análisis de los resultados finales y conclusiones

Una vez se ha generado el modelo final, llega el momento de evaluar su rendimiento y estimar su error de generalización. Para ello, se utilizará por primera vez el conjunto de *test* que se separó al principio. Cabe añadir que los datos de este conjunto deben ser normalizados según los parámetros obtenidos al normalizar el conjunto de entrenamiento. Para esta tarea, se calculará la métrica de error sobre los datos de *test* y se mostrarán la curva ROC y la matriz de confusión para un análisis más detallado del comportamiento del modelo.

3.1. Métricas

Tras ejecutar el modelo sobre los datos de prueba, se obtienen los siguientes resultados:

Modelo	Test Loss	Test Accuracy
Modelo Final	0.2654	0.9459

Cuadro 1: Resultados del modelo final sobre el conjunto de *test*

3.2. Curva ROC

Se trata de una gráfica que muestra la tasa de falsos positivos (eje X) contra la tasa de verdaderos positivos (eje Y) para varios valores candidatos del umbral de discriminación (valor a partir del cual decidimos que un caso es un positivo) entre 0 y 1. Acerca de su interpretación, cabe decir que en modelos con buen rendimiento, el área bajo la curva tiende a ser mayor (la curva crece muy rápidamente al principio y se mantiene "constante" a lo largo del eje X). De hecho, es más o menos el comportamiento que se observa en la figura 8.

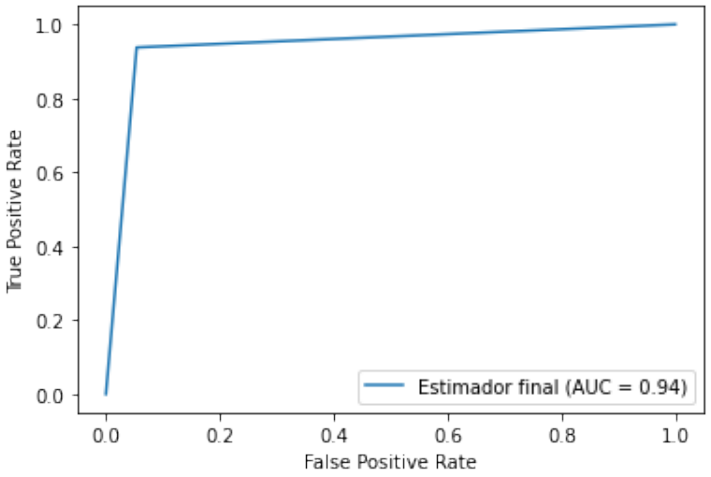


Figura 8: Curva ROC

3.3. Matriz de confusión

La matriz de confusión es útil para observar que clases confunde, válgase la redundancia, con mayor frecuencia entre sí. Permite observar el número de falsos/verdaderos positivos/negativos.

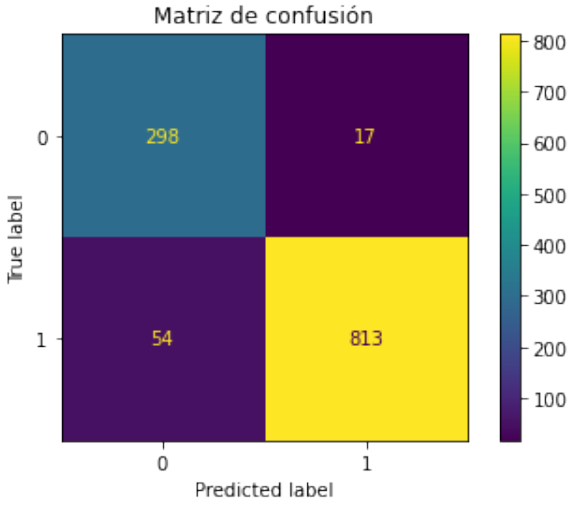


Figura 9: Matriz de confusion

3.4. Conclusiones

Tal y como muestra la tabla 1, el modelo alcanza un *accuracy* del 94'59 % en el conjunto de *test*, (teniendo un error de generalización aproximado del 5'41 %). Se trata de una precisión bastante alta (en torno al 5 %), por lo que el ajuste del modelo ha sido exitoso. Un detalle importante es que se trata de un valor bastante similar al *accuracy* del conjunto de validación, lo cual indica la ausencia de sobreajuste (*overfitting*), tratándose así de un modelo aparentemente bastante estable y robusto.

En general, se podría considerar que, en el ámbito sanitario, es preferible que un modelo produzca un falso positivo a un falso negativo, lo cual ocurre en el modelo. Aun así, debería de ser un experto quien valorase este aspecto. En cualquier caso, tanto el número de falsos positivos como falso negativos es relativamente bajo.

Por otro lado, es destacable que, tras haber partido de una precisión inicial de aproximadamente un 80 % (en validación), se ha logrado alcanzar más de un 94 %, por lo que se puede considerar que se ha conseguido ajustar la red al problema concreto y aplicar las mejoras adecuadas.

Referencias

[Dataset] CoronaHack-Chest X-Ray-Dataset, *Classify the X Ray image which is having pneumonia*
https://www.kaggle.com/praveengovi/coronahack-chest-xraydataset

[MiniBatch] Yoshua Bengio (Cornell University), *Practical recommendations for gradient-based training of deep architectures*
https://arxiv.org/abs/1206.5533

[OpenCV] Docs OpenCV, *OpenCV*
https://docs.opencv.org/

[TensorFlow] TensforFlow, *TensforFlow*
https://www.tensorflow.org/?hl=es-419

[Early Stopping] Towards Data Science, *A practical introduction to early stopping in machine learning*
https://towardsdatascience.com/a-practical-introduction-to-early-stopping-in-machine-learning-550ac88bc8fd

[Kras.io] Transfer learning & fine-tuning, *Transfer learning & fine-tuning*
https://keras.io/guides/transfer_learning/#the-typical-transferlearning-workflow

[Towards Data Science] Cody Glickman, *Data Augmentation in Medical Images*
https://towardsdatascience.com/data-augmentation-in-medical-images-95c774e6eaae