Universidad de Granada

Visión por computación

Clasificación de imágenes usando CNNs

Francisco Luque Sánchez
María del Mar Ruiz Martín

16 de enero de 2018

Índice

1. Introducción

En esta práctica se tratará el problema de la clasificación de objetos en imágenes, utilizando concretamente redes neuronales convolucionales (*CNNs*). El problema que se abordará consiste en tratar de distinguir perros de gatos utilizando estos modelos. Se comenzará con un modelo simple, el cual se irá modificando para tratar de mejorar su capacidad para clasificar.

1.1. Conjunto de datos utilizado

El conjunto de datos utilizado se ha generado utilizando las bases de datos mostradas en [1, 2, 3]. Se han extraído todas las imágenes de las mismas y etiquetado en dos clases (perros y gatos), obteniéndose un conjunto total de unos 13000 gatos y 25000 perros. Dicho conjunto se ha dividido en dos subconjuntos, un conjunto de entrenamiento (unos 25500 ejemplos) y uno de test (en torno a 12500 ejemplos), tratando de mantener la proporción de perros y gatos lo más parecida posible en ambos conjuntos.

En cuanto al tamaño de las imágenes utilizado, se han redimensionado todas ellas a un tamaño de 64×64 píxeles con codificación RGB (es decir, se trabajará con imágenes de entrada de tamaño $(64, 64, 3)$. Se utilizan imágenes de tan pequeño tamaño porque el uso de imágenes de mayor tamaño provoca un aumento de tamaño muy notable de la red neuronal utilizada, lo que se traduce en un aumento del tiempo de cómputo muy considerable. Además, se comenzó haciendo una prueba con imágenes de tamaño 128×128, y la diferencia en los resultados obtenidos no era significativa. Finalmente, este trabajo tiene la finalidad de estudiar las diferencias de capacidad de clasificación de las redes neuronales convolucionales en función a su estructura y parámetros, por lo que en principio no necesitamos crear ejemplos muy potentes que permitan una clasificación muy precisa. Se decide por tanto utilizar imágenes de este tamaño, aunque pueda provocar que en fases finales del trabajo se pierda un poco de capacidad de predicción al no utilizar imágenes de tamaño mayor.

1.2. Aspectos de implementación

Todo el código se ha desarrollado utilizando el *framework* TensorFlow [4], que es una librería de código abierto desarrollada por Google, orientada a la implementación de soluciones utilizando inteligencia artificial. Esta librería permite la definición de forma sencilla de estructuras de redes neuronales de varios tipos, entre ellas redes neuronales convolucionales, que es el tipo de redes en las que se centra el trabajo. Además, permite especificar los recursos del equipo que se destinan a cómputo, dando gran flexibilidad al programador a la hora de hacer los experimentos. El primer modelo desarrollado, en particular, se ha hecho utilizando un tutorial de la documentación del *framework*, que se puede consultar en [5].

El código se ha estructurado en 5 archivos distintos para cada modelo. En el archivo `model.py` se establece la estructura de la red neuronal. En los archivos `model_train.py` y `model_test.py` se establece la ejecución de las operaciones de entrenamiento y test. En el archivo `input.py` se implementan las funciones de lectura de imágenes desde archivo. Finalmente, el archivo `predict_image.py` permite que se pase un nombre de imagen como argumento, y se utiliza el modelo entrenado para predecir si en dicha imagen hay un perro o un gato.

2. Modelos implementados

2.1. Model básico (`base_model`)

Comenzamos describiendo el primer modelo implementado. Es el modelo más simple de los estudiados. Pasamos a ver su estructura.

2.1.1. Estructura de la red

La primera de las redes se organiza de la siguiente manera. Recibe un batch de 128 imágenes de tamaño $64 \times 64 \times 3$. La primera operación que realiza consiste en una capa de convolución que extrae 64 filtros para cada una de las imágenes. Después, tiene una capa de pool que reduce el tamaño de las imágenes a la mitad, por lo que tras esta capa se tiene un conjunto de imágenes de tamaño $128 \times 32 \times 32 \times 64$. Tras esto, se redimensiona la capa para que tenga una forma de 128×65536 características, y se pasa dicho vector a una capa completamente conectada de tamaño 65536×348. Esta capa completamente conectada se conecta a otra capa completamente conectada con dos neuronas, que nos darán la probabilidad de que el elemento pertenezca a la clase gato (neurona 0) o la clase perro (neurona 1). Las funciones de activación de todas las capas es la función *Relu*. El esquema de la misma es el siguiente:

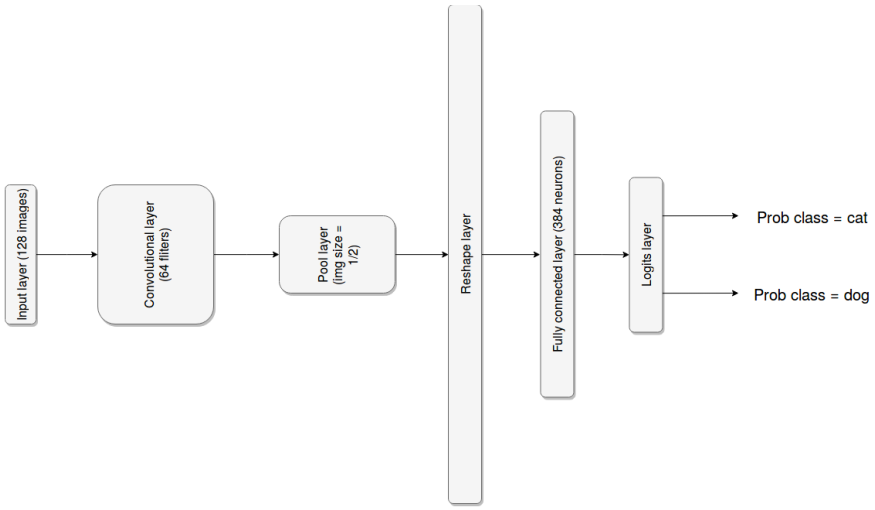


Figura 1: Esquema del modelo simple de red neuronal convolucional

2.1.2. Aprendizaje de la red

Una vez definida la estructura de la red, vamos a explicar ligeramente el funcionamiento del aprendizaje de la misma. En cada etapa del aprendizaje, se genera aleatoriamente del conjunto de imágenes de entrenamiento un subconjunto de 128 ejemplos. La red neuronal clasifica dichas imágenes, otorgando una probabilidad a cada una de ellas de que pertenezca a una clase o a la otra. Una vez hecha la clasificación, se calcula el error cometido como la media de los errores cometidos. El error cometido es la diferencia entre la probabilidad estimada para cada clase y la clase real de la imagen. Una vez calculado dicho error, se propaga con un gradiente descendente para modificar los pesos de todas las capas. La tasa de aprendizaje es constante durante todo el entrenamiento, con un valor de 0,001. Se realizan 8000 pasos de aprendizaje.

2.1.3. Evaluación del aprendizaje durante el entrenamiento

Para ir viendo cómo evoluciona el aprendizaje de la red durante el entrenamiento, se ejecuta simultáneamente un test sobre la red neuronal. Este test coge un conjunto aleatorio de 10000 imágenes del conjunto de test y lo clasifica utilizando la red neuronal. Una vez clasificadas las mismas (se obtiene la probabilidad de que cada una de las imágenes pertenezca a una de las dos clases, y se devuelve como clase la que tenga una probabilidad mayor), se hace el cociente entre el total de ejemplos bien clasificados y el número de ejemplos totales, obteniéndose así el tanto por uno de instancias bien clasificadas.

2.1.4. Gráficas de resumen del aprendizaje

Utilizando una de las utilidades de *TensorFlow*, llamada *TensorBoard*, se han creado varios gráficos que permiten ver cómo evoluciona la capacidad de predicción de la red en función al número de iteraciones de aprendizaje que ha realizado la misma.

Nos centraremos principalmente en dos gráficas. Una que muestra la evolución de la función de pérdida, la cual tratamos de minimizar, y otra que muestra la capacidad de predicción sobre el conjunto de test.

Estas dos gráficas van a ser nuestra principal forma de obtener información sobre la evolución del aprendizaje de la red neuronal. Una de las ventajas que ofrece TensorFlow es que permite al programador establecer, de forma muy simplificada (tiene implementado un sistema de clases para realizar todo el trabajo), qué variables se quieren supervisar durante el entrenamiento. De esta manera, podemos colocar supervisores que mantengan un historial de los pesos de las distintas capas de la red, de si ciertas neuronas se activan o se inhiben para una determinada foto, cómo se propagan los gradientes por la red neuronal con el algoritmo de *backpropagation*... y mostrar toda esta información a posteriori con una interfaz web (es esta interfaz la que se conoce como *TensorBoard*). No obstante, en este trabajo no se mostrará la capacidad completa de esta herramienta, ya que su configuración es tediosa, y aunque no ralentiza en exceso el sistema, sí que se puede apreciar cierta pérdida de rendimiento. Además, la interpretación de estos gráficos, a pesar de que se muestran de forma bastante clara, no es nada sencilla, y requieren de un conocimiento previo sobre redes neuronales bastante profundo. Creemos por tanto que la extracción e interpretación de dichas gráficas queda fuera de las competencias que intenta cubrir el trabajo, y por esto mostramos sólo las gráficas cuya interpretación aporta información realmente relevante sobre el aprendizaje de la misma.

Pasamos entonces a ver la gráfica de la evolución de la función de pérdida en los distintos pasos de entrenamiento:

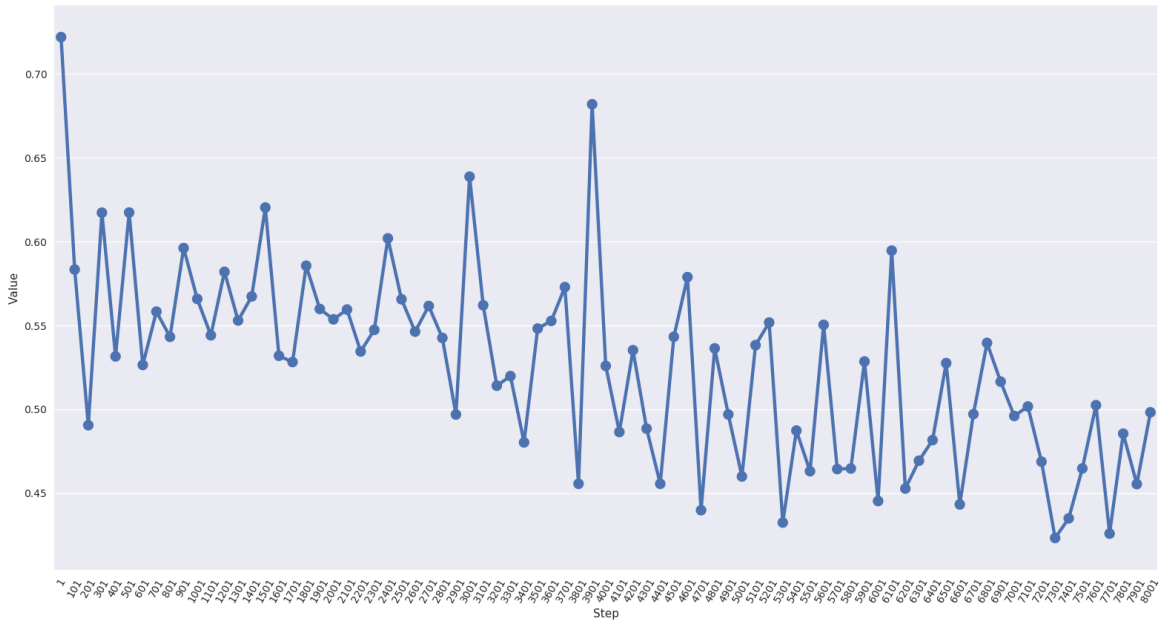


Figura 2: Evolución de la función de pérdida

Podemos observar cómo la función de pérdida tiene un comportamiento un poco caótico en este caso, aunque sí que se observa cierta tendencia al decrecimiento. Se comienza con un valor cercano a 0.7, el cual se consigue decrementar hasta 0.43 en la iteración 7700. No obstante, se ve que el comportamiento no es del todo bueno, ya que se dan muchos pasos hacia atrás durante el entrenamiento. Pasamos ahora a ver el comportamiento de la gráfica que muestra la evolución la precisión del modelo:

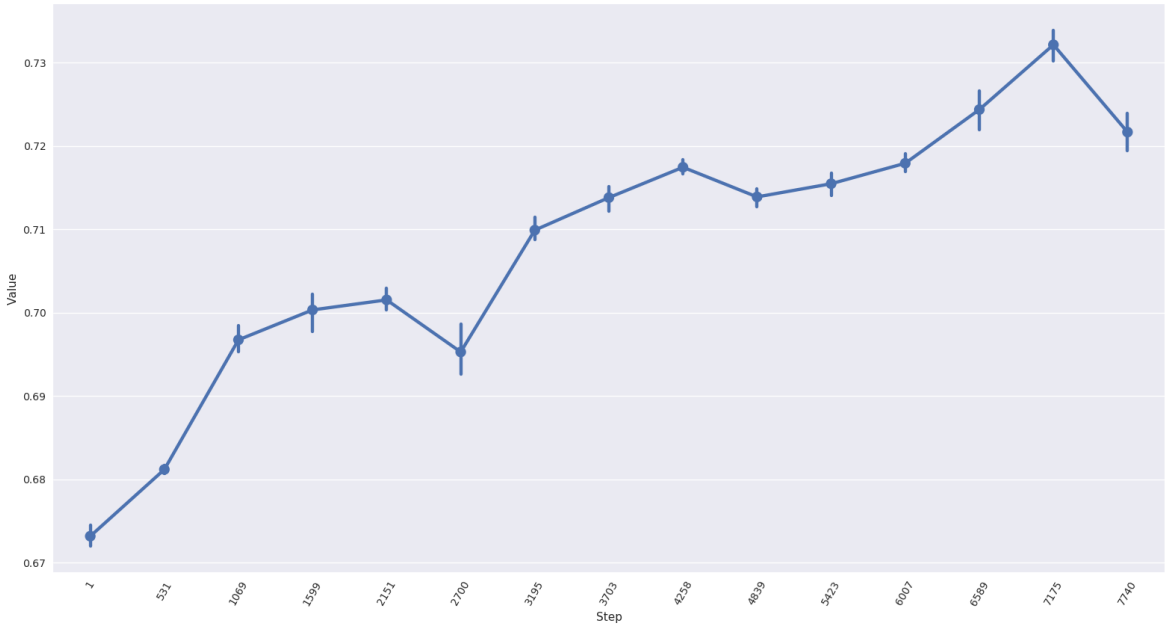


Figura 3: Evolución de la precisión de la red neuronal

Se puede observar una clara tendencia a la mejora durante todo el proceso de aprendizaje, aunque no se una mejora muy significativa. El sistema comienza con una capacidad de predicción de en torno al 67 % de efectividad, y consigue mejorar dicha capacidad de predicción hasta alcanzar una precisión de algo más del 73 %. No obstante, estos resultados no son del todo satisfactorios, ya que hemos construido un modelo muy básico y todavía clasificamos mal algo más de un cuarto de las instancias que tenemos. En el resto de la práctica se irán proponiendo mejoras a este modelo básico, que tratarán de mejorar su capacidad de predicción. La primera de las modificaciones consistirá simplemente en introducir un *Weight Decay* sobre los pesos de la red neuronal. Pasamos a describirlo y ver los resultados en la siguiente sección.

2.2. Primera modificación: *Weight Decay* (wd_model)

Como ya hemos dicho al final de la sección anterior, la primera modificación que vamos a introducir es lo que se conoce como *Weight Decay*, o disminución de pesos. Esta modificación consiste en añadir una penalización a los pesos de la red en la función de pérdida. Concretamente, se añade un término por cada capa de la red en las que se aplica, correspondiente a la norma L_2 del vector de pesos, multiplicado por un parámetro a configurar por el usuario. Esto hace que cada vez que se propaga el error, además de modificar los pesos en función de la capacidad de predicción, se reduce la norma del vector, haciendo que los pesos no aumenten indefinidamente. Veremos que esta modificación nos servirá para que la red tenga un comportamiento mucho mejor que en el modelo anterior.

En cuanto a la estructura de la red y el esquema de aprendizaje de la misma, no comentaremos nada más, ya que es exactamente la misma que la que teníamos para el ejemplo anterior. De nuevo tendremos una capa de convolución, una capa de reducción de tamaño, una de alisamiento de la dimensión de las imágenes para convertirlas en vectores, una capa completamente conectada y la salida con dos neuronas. La única modificación interesante es que los pesos de todas las capas llevan asociado el *Weight Decay*, lo que se verá claramente en los valores de la función de pérdida, que tomará valores mucho mayores (recordemos que en el ejemplo anterior, el valor de la función no sobrepasaba el valor 1, mientras que ahora comenzará en un valor cercano a 600). El esquema de aprendizaje es el mismo nuevamente, utilizándose un esquema de gradiente descendente con una tasa de aprendizaje constante para propagar el error.

Lo que sí tendremos ahora es un aprendizaje mucho más largo. Mostraremos dos ejecuciones distintas. La primera es una ejecución en la que se realizan 10000 etapas de aprendizaje. Tras la ejecución, se observó que, aunque se había mejorado el resultado de la ejecución básica, la red neuronal no había presentado todavía un estancamiento en los resultados, por lo que se decide modificar este parámetro, y establecer el límite de etapas de aprendizaje en 100000. Esta estimación es muy exagerada, ya que probablemente se produzca antes un estancamiento, pero como las ejecuciones son costosas, se decide poner un número desproporcionado de pasos, para cortar el proceso cuando se observe un estancamiento en la disminución de la función de pérdida. Más adelante en la práctica se introducirá un método que permita automatizar el momento del entrenamiento en el que se corta el aprendizaje. Esto es lo que se llama *Early Stopping*, y básicamente se basa en hacer que se pare el entrenamiento cuando se produce un estancamiento en la mejora de la función de pérdida, esto es, cuando tras un número significativo de iteraciones no se consigue disminuir el valor de la misma lo suficiente, o no llega a mejorarse.

Pasamos a ver los resultados de las ejecuciones de entrenamiento conseguidos por esta red neuronal.

2.2.1. Gráficas de resumen del aprendizaje

Vamos a ver, como hemos dicho antes, las gráficas de dos ejecuciones del aprendizaje para este modelo. La primera que se muestra es una primera ejecución, en la que se establecieron 10000 pasos de aprendizaje. Tras ver que al finalizar la ejecución la red neuronal no se había estancado, se realiza otra ejecución de 100000 pasos para ver hasta qué punto la red neuronal es capaz de mejorar las predicciones. La primera gráfica es la siguiente:

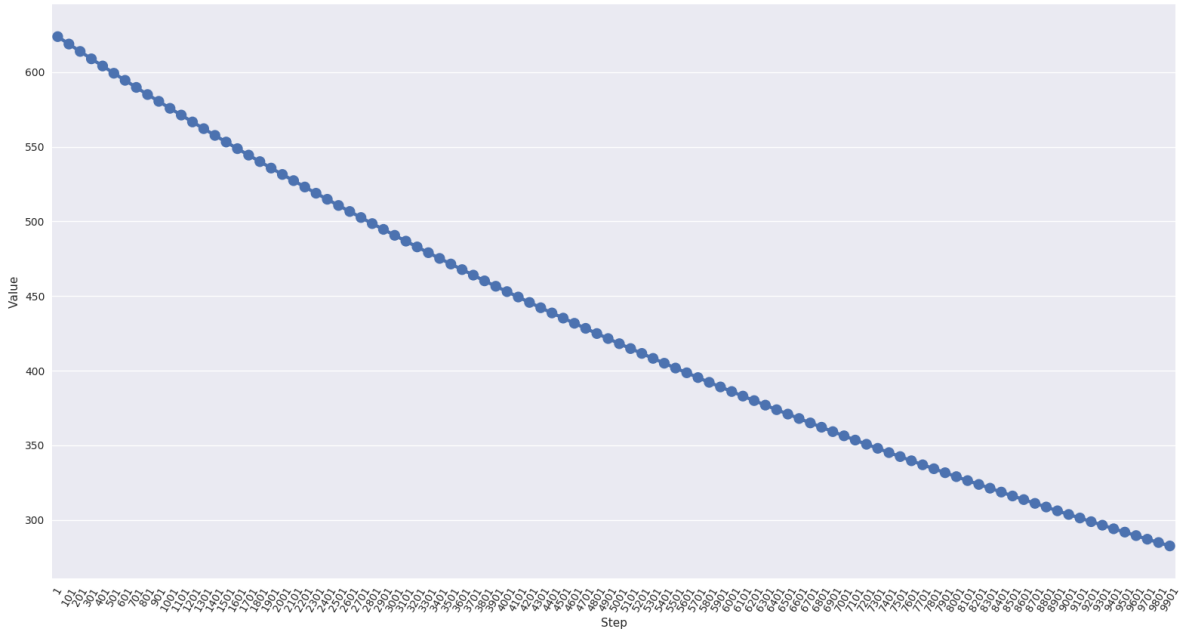


Figura 4: Evolución de la función de pérdida

Claramente, la función de pérdida no se ha estabilizado. Si observamos la gráfica de la capacidad de predicción, tenemos que:

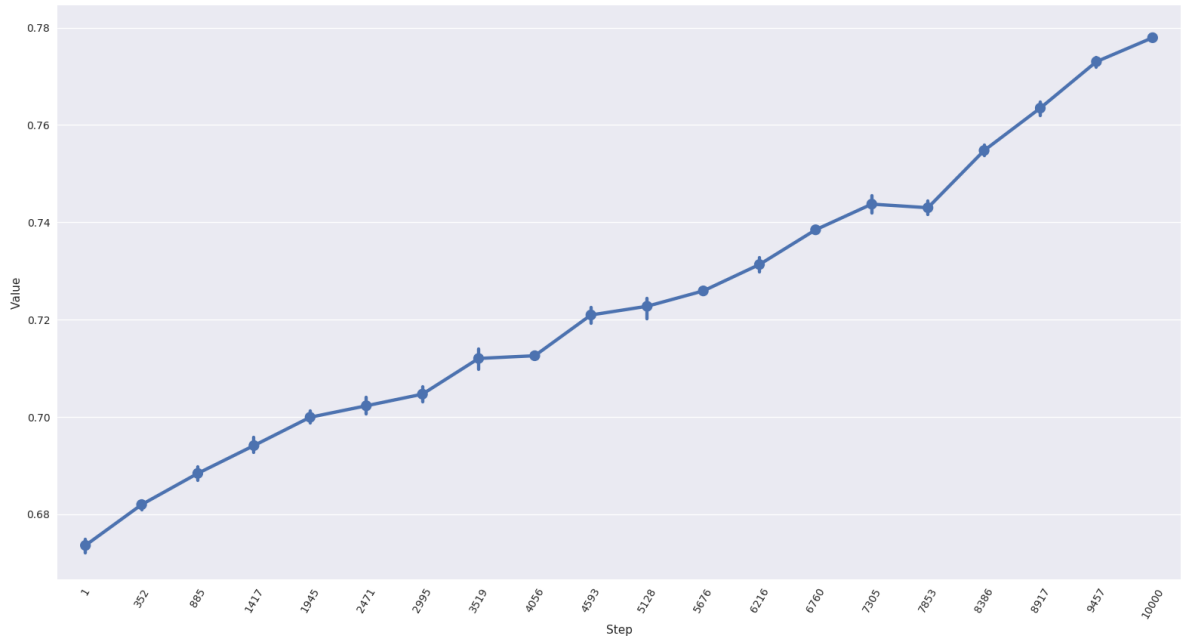


Figura 5: Evolución de la capacidad de predicción

Donde hemos conseguido un porcentaje de acierto cercano al 78 %. No obstante, se ve claramente en ambas gráficas que el resultado es claramente mejorable. En la segunda ejecución, la gráfica de la función de pérdida es la siguiente:

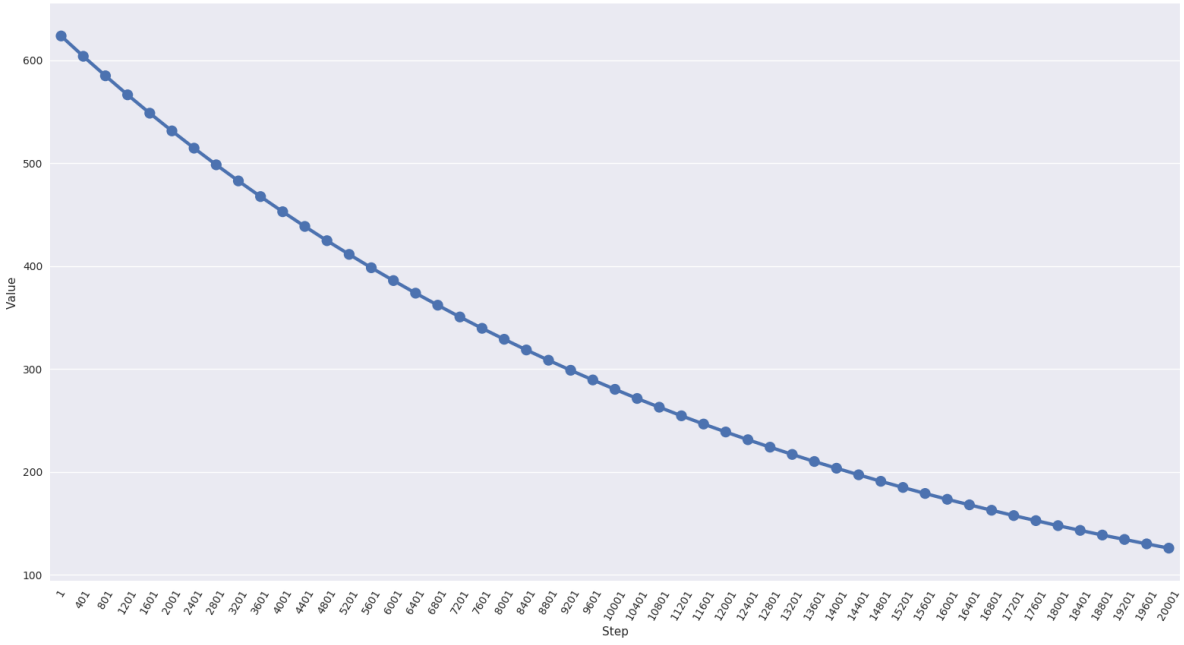


Figura 6: Evolución de la función de pérdida

Como se puede ver en la gráfica, ahora se han ejecutado 20000 pasos de aprendizaje. A pesar de ello, todavía no se ha estabilizado la función de pérdida, pero como observaremos en la gráfica

siguiente sí que se ha estabilizado la capacidad de predicción en la red:

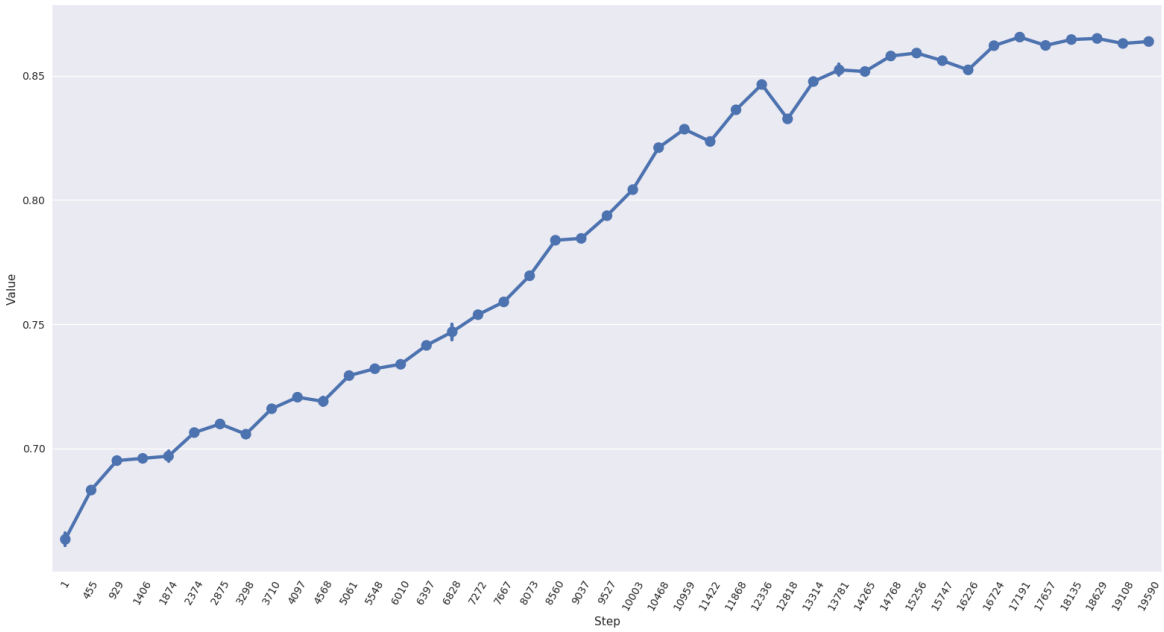


Figura 7: Evolución de la capacidad de predicción

Podemos ver en la gráfica que en los últimos 5000 pasos apenas se ha conseguido mejorar el porcentaje de clasificación del modelo. No obstante, con esta red hemos conseguido mejorar la capacidad de aprendizaje de la misma, subiendo el porcentaje de acierto en clasificación de un 73 % a un 86,5 %. Esta mejora ha supuesto, si tenemos en cuenta el error (27 % frente a 13,5 %), conseguir clasificar correctamente, en media, la mitad de los ejemplos que antes no éramos capaces de clasificar. Supone por tanto una mejora muy importante en el modelo. Además, hemos conseguido ganar mucha estabilidad a la hora de la mejora del modelo, ya que como hemos podido observar, la función de pérdida ha ido decrementando su valor en todo momento, al contrario de lo que ocurría en el primer modelo, en el que teníamos oscilaciones muy fuertes. Gran parte de la culpa de este fenómeno es de los términos que hemos añadido, ya que el tener en cuenta como penalización la norma de los pesos de la red, y decrementarlos todos en cada paso de aprendizaje en función de su norma, es normal que la función de pérdida no tienda a aumentar, ya que la parte correspondiente a los pesos será mucho más significativa que la correspondiente a la mala clasificación. No obstante, es innegable que la implementación de *Weight Decay* ha conseguido mejorar enormemente los resultados obtenidos por la red, a la vista de los resultados obtenidos.

2.3. Segunda modificación: *Variable Learning Rate* (vlr_wd_model)

En vista de la mejora que nos supone el modelo anterior, trabajaremos sobre éste y le añadiremos sucesivas modificaciones con fin de estudiar la influencia de cada una en nuestro modelo. Hasta ahora habíamos trabajado con una tasa de aprendizaje fija. Este parámetro resulta difícil de ajustar, puesto que tenemos que encontrar un equilibrio entre la capacidad de explorar el espacio de soluciones que nos proporciona un valor de tasa de aprendizaje alto, y la capacidad de

optimizar los resultados en un entorno pequeño que conseguiríamos con una valor bajo para la tasa.

Para solventar esta dificultad de ajustar el parámetro, y conseguir una mayor flexibilidad entre la explotación y la exploración del espacio de soluciones, utilizaremos una tasa de aprendizaje variable. De esta forma se comienza con un valor de tasa de aprendizaje relativamente alto, que nos permita realizar al inicio un buen proceso de exploración, mientras este parámetro se reduce gradualmente, dando así paulatinamente más importancia al proceso de explotación.

Como ya se ha comentado, se utiliza la misma topología para nuestra red neuronal que la que tenía el modelo anterior, utilizando también el mismo esquema de aprendizaje con gradiente descendente, pero en esta ocasión con tasa de aprendizaje variable.

A continuación vamos a estudiar los resultados obtenidos en la ejecución del nuevo modelo.

2.3.1. Gráficas de resumen del aprendizaje

De forma análoga a como se ha procedido hasta ahora, mostraremos las gráficas sobre la función de pérdida y la capacidad de predicción del modelo a lo largo de su fase de entrenamiento. En primer lugar mostramos la función de pérdida:

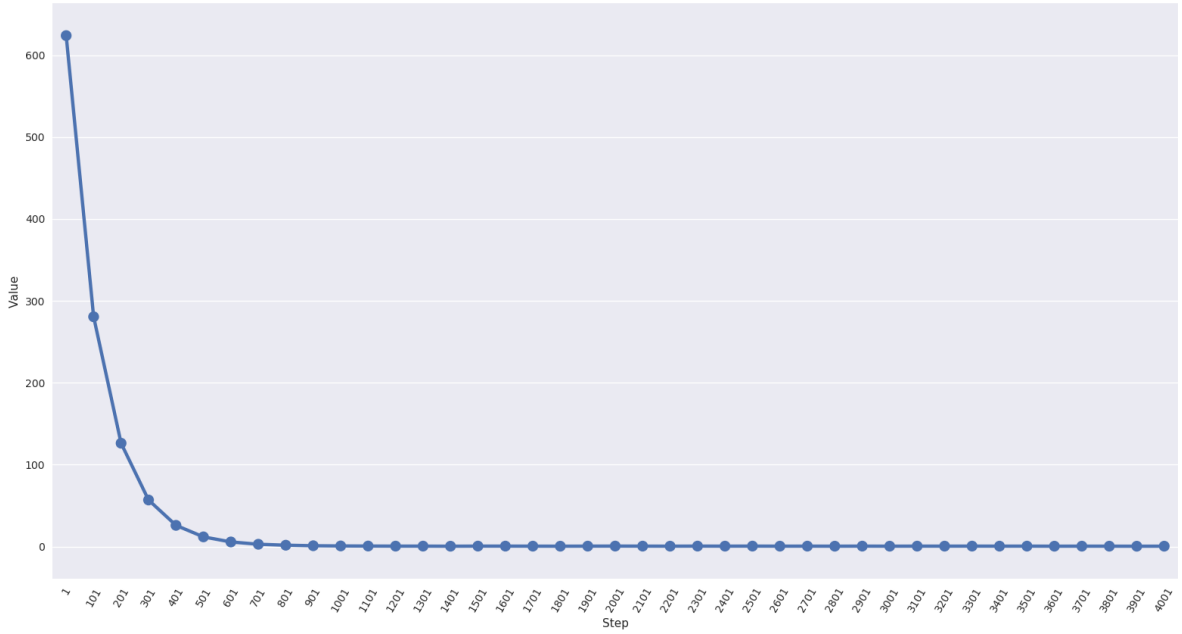


Figura 8: Evolución de la función de pérdida

Encontramos por un lado que la disminución de la función de pérdida es mucho más rápida que la que obteníamos en el modelo anterior, bajando muy rápidamente a valores inferiores a 10 y llegando por debajo de la unidad, cuando el modelo anterior no era capaz obtener valores por debajo de 100. Veamos a continuación como evoluciona la capacidad de predicción del modelo actual.

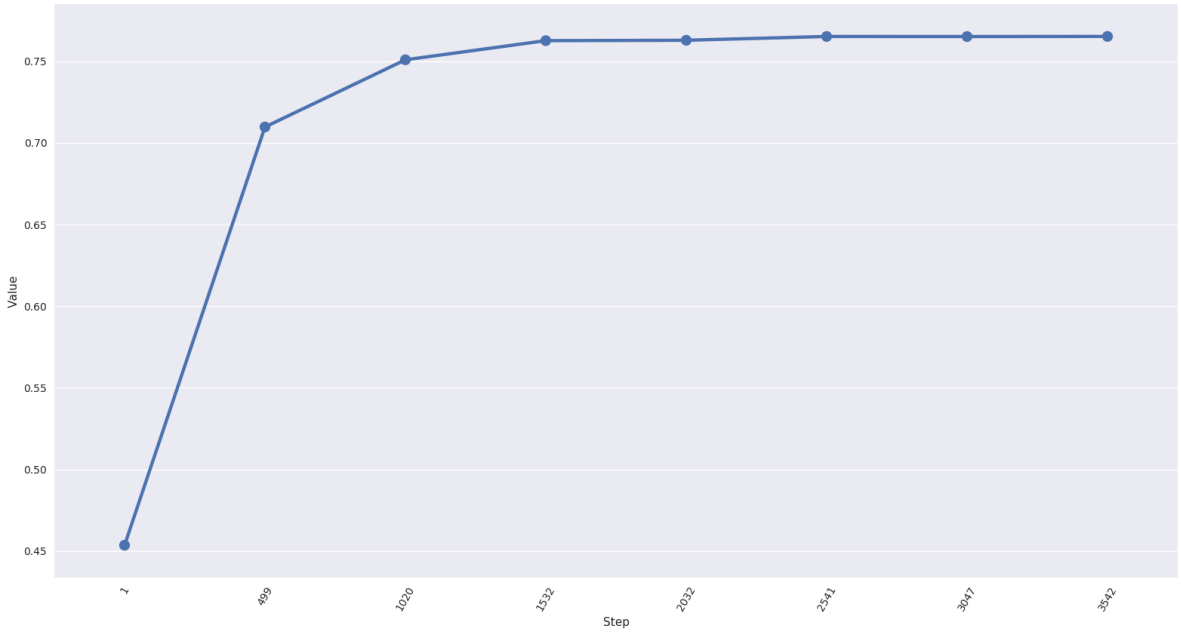


Figura 9: Evolución de la capacidad de predicción

En esta ocasión encontramos que aumenta muy rápidamente, pero pronto se estanca, no consiguiendo alcanzar una precisión superior al 76 %. En vista de las dos gráficas anteriores, y del comportamiento que obteníamos en el modelo anterior, podemos deducir que nuestro modelo se centra demasiado en reducir el valor de pérdida provocado por la penalización aplicada por el *Weight Decay* en lugar de mejorar la solución a nuestro problema.

2.4. Tercera modificación: *Variable Learning Rate* sin *Weight Decay* (vlr_model)

Una vez estudiado el modelo anterior, nos cabe preguntarnos si el uso de una tasa de aprendizaje variable nos puede ayudar en aquellos casos en los que no apliquemos *Weight Decay*, o lo apliquemos en una menor medida.

Con el fin de comprobar esta idea, creamos un nuevo modelo idéntico al anterior con la salvedad de que tan solo aplicaremos *Weight Decay* en la capa densa. Nuestro propósito es comprobar que al no haber tanta penalización en la función de pérdida, nuestra red neuronal convolucional podrá realizar una mejor clasificación.

Pasámos a comprobar los resultados obtenidos por el tercer modelo:

2.4.1. Gráficas de resumen del aprendizaje

De nuevo pasamos a mostrar en primer lugar la evolución de la función de pérdida, para a continuación estudiar la capacidad de predicción que nos aporta el modelo.

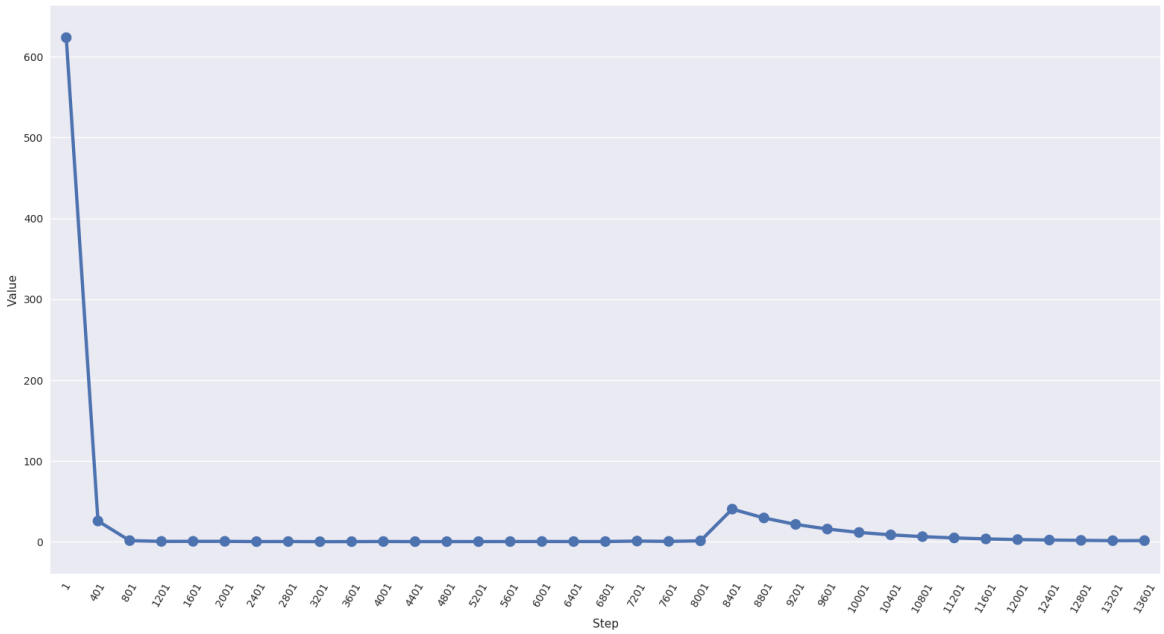


Figura 10: Evolución de la función de pérdida

De nuevo encontramos que la función de pérdida, a pesar de comenzar con valores relativamente altos, rápidamente decrece y alcanza valores muy pequeños y se estabiliza. Sin embargo, encontramos que llega un punto en el que toma un alto valor. A pesar de que tras el salto vuelve a decrecer, lo hará de forma muy lenta y no llega a alcanzar los valores en los que se situaba antes. Veamos cómo se comporta en la predicción:

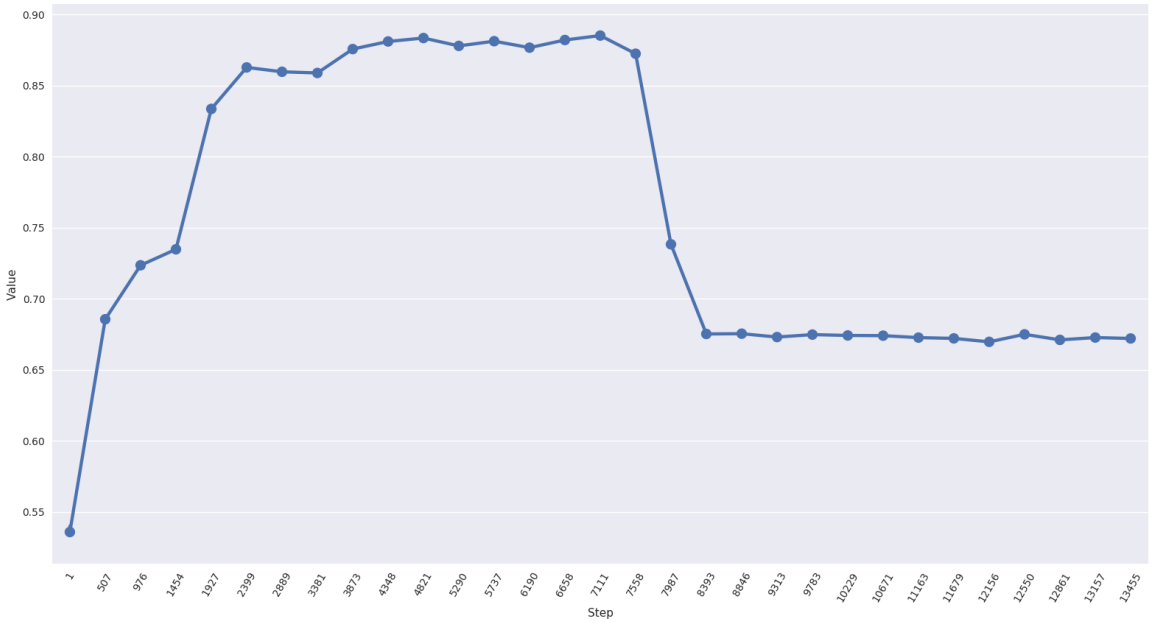


Figura 11: Evolución de la capacidad de predicción

En esta ocasión encontramos, a diferencia del modelo anterior, que tras una gran subida inicial, sigue creciendo, aunque de forma mucho más lenta. A pesar de comenzar con una tasa de predicción por debajo del 55 %, este modelo consigue alcanzar un 89 % de acierto, siendo notablemente mejor que todo lo que habíamos conseguido hasta ahora.

Sin embargo, encontramos un rápido descenso coincidiendo con el punto en el que la función de pérdida aumentaba. La capacidad de predicción del modelo se reduce drásticamente estabilizándose en torno a un 66 %, valor muy inferior. Esto nos hace plantearnos que debemos establecer un criterio de parada, para que la red neuronal finalice su entrenamiento y no se lleguen a estas situaciones.

2.5. Cuarta modificación: *Normalización* (norm_model)

Puesto que el modelo anterior nos arrojaba realmente buenos resultados, nos proponemos quedarnos con él y añadirle una modificación en búsqueda de unos mejores resultados. Sabemos que, en general, los clasificadores y los regresores trabajan mejor con datos normalizados. Nos proponemos por tanto añadir una capa de normalización a nuestro modelo.

Tomaremos la normalización *local_response_normalization* implementada en *TensorFlow*. Puesto que trabajamos con datos de 4 dimensiones, el normalizador supondrá que trabaja con arrays tridimensionales, en los que cada elemento es un vector unidimensional. Sera sobre cada uno de estos vectores sobre los que se aplicará la normalización.

2.5.1. Estructura de la red

La arquitectura que tendremos en este modelo será idéntica a la básica, pero precisaremos de una capa de normalización tras la capa de *pool*. Queda por tanto la topología de nuestra red neuronal como sigue:

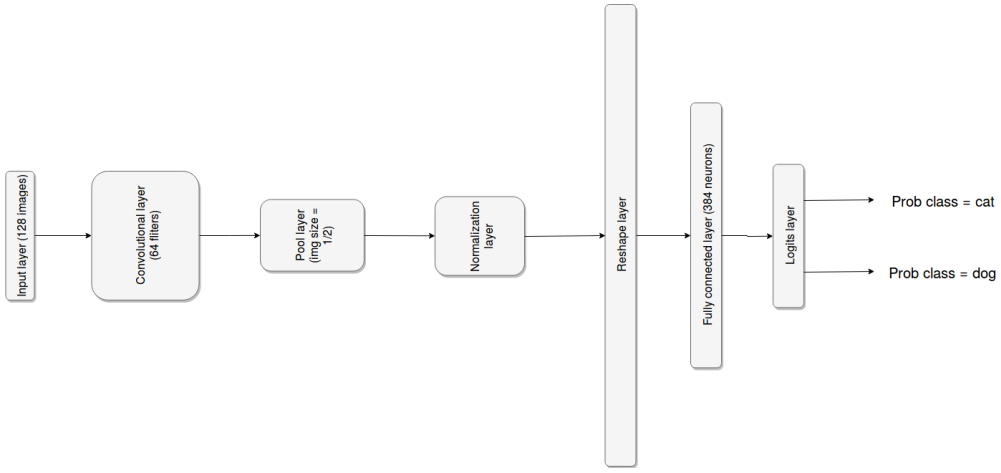


Figura 12: Esquema del modelo con normalización de red neuronal convolucional

2.5.2. Gráficas de resumen del aprendizaje

Vamos a continuación a estudiar los datos arrojados por nuestro modelo.

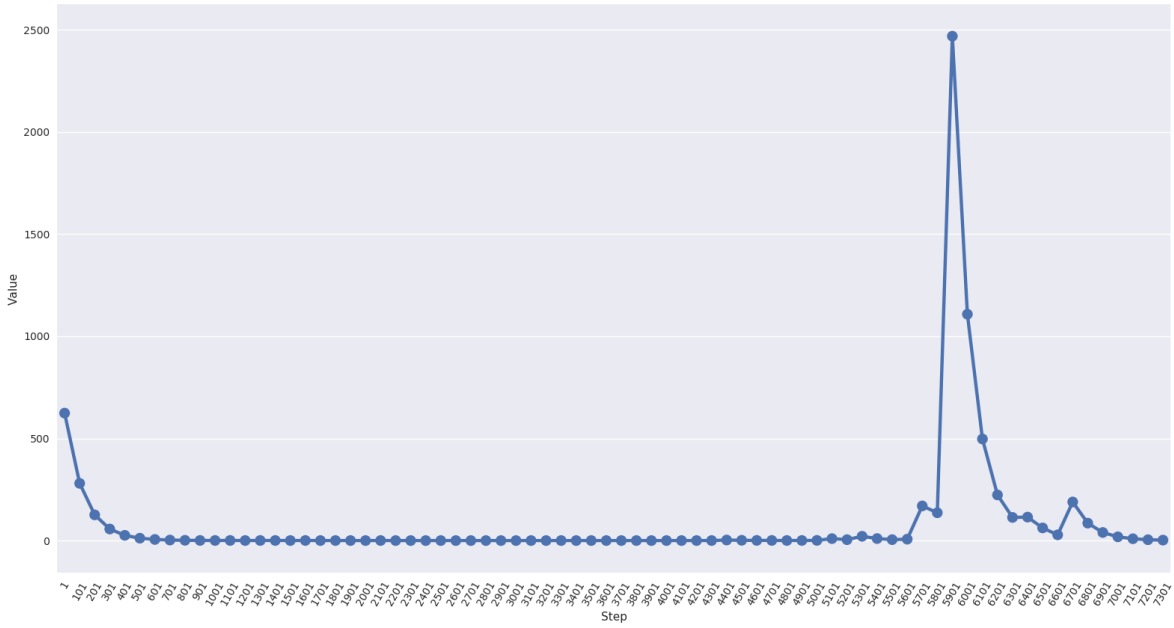


Figura 13: Evolución de la función de pérdida

Encontramos que la red presenta un buen comportamiento, ya que baja rápidamente el valor de la pérdida y consigue mantenerlo estable siempre con tendencia a decrecer. Sin embargo, encontramos un punto en el cual alcanza un valor considerablemente alto. A partir de este punto consigue reducir su valor pero quedando siempre por encima de los valores en los que se encontraba antes.

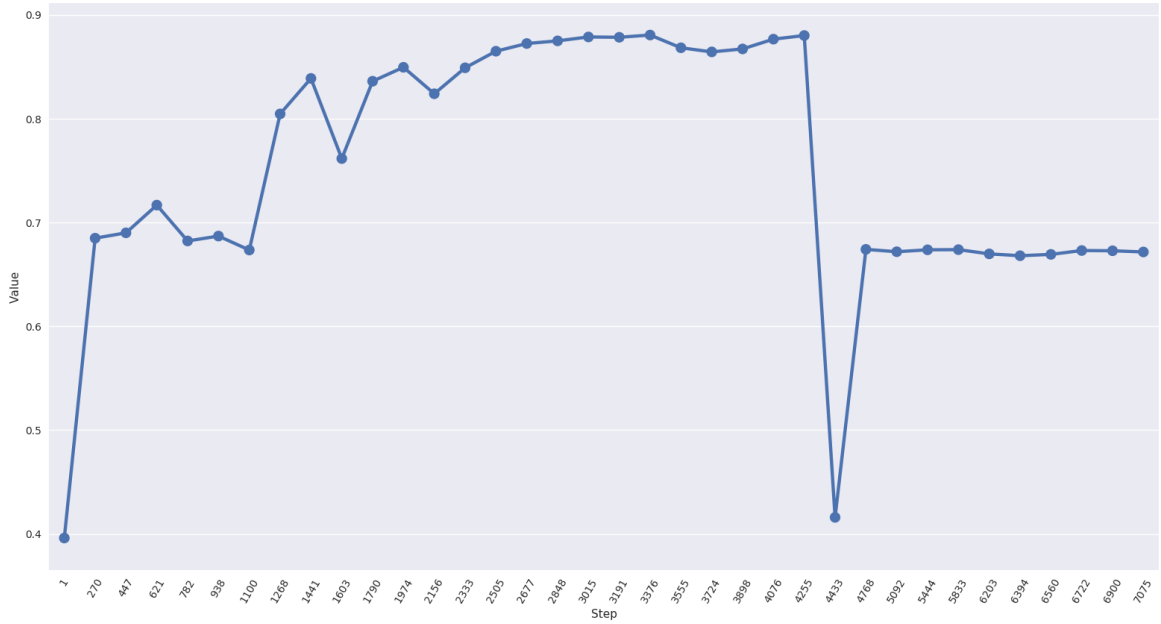


Figura 14: Evolución de la capacidad de predicción

A pesar de que la clasificación empieza realmente mal, con menos de un 40 % de clasificación, la red neuronal consigue mejorar poco a poco y estabilizarse en valores relativamente buenos, moviéndose

entre los pasos 2200 y 4200 por encima del 86 % de acierto. Sin embargo, encontramos un punto en el que la capacidad de predicción baja notablemente, alcanzando casi su valor mínimo y, aunque consigue mejorar, queda muy lejos de los resultados que anteriormente alcanzaba.

A pesar de que este modelo también a demostrado alcanzar un buen funcionamiento, la máxima capacidad de predicción ha sido de 88.2 %, ligeramente inferior a la que conseguíamos en el modelo anterior. Además, la aplicación de la normalización ralentiza considerablemente el proceso de aprendizaje. Por tanto, en principio descartaremos la aplicación de la capa de normalización.

2.6. Quinta modificación: dropout + *early stopping* (dropout_model)

Vamos ahora a introducir dos modificaciones sobre el modelo básico. En primer lugar, se va a introducir una pequeña modificación a la estructura de la red. Realmente esta modificación no supone un cambio notable en la topología de la misma, aunque en TensorFlow se implementa como una capa más dentro de la red neuronal. Nos referimos a una técnica que se conoce como *dropout*. Esta técnica consiste en, mientras se está en la fase de entrenamiento, desactivar la salida de algunas neuronas (hacer que su salida sea 0), para evitar que la capacidad de clasificación de la red se concentre en unas pocas de neuronas exclusivamente. Lo que se hace es colocar una capa de tipo `dropout` tras la capa de la red a la que se quiere aplicar esta modificación. Esta capa acepta como parámetros una capa de una red neuronal (a fin de cuentas, esto es un vector de pesos), y un parámetro entre 0 y 1, que es la probabilidad de conservar una determinada neurona. En función a ese parámetro, lo que se hace es generar un número aleatorio para cada una de las neuronas de la red, que decidirá si dicha neurona está activa o inactiva en ese paso del aprendizaje. En caso de que la neurona haya de quedar activa, se multiplica el valor de salida de la misma por $\frac{1}{p}$, donde p es la probabilidad de conservar a la neurona. Si la neurona ha de quedar inactiva, se reemplaza su valor por 0. La multiplicación se realiza para que la norma del vector de salida de la red neuronal sea similar a la que tendría en caso de tener todas las neuronas activas. Durante la evaluación de la red neuronal, esta capa no se ejecuta, ya que se quiere que todas las neuronas participen en el proceso de clasificación de imágenes al final.

La idea de utilizar esta técnica apareció hace relativamente poco tiempo (según [6], fue propuesta en 2012, y detallada en 2014). El argumento que sustenta esta técnica es que, usualmente, los equipos de trabajo funcionan mejor cuando la carga se reparte de forma más o menos equitativa entre todos los miembros que lo conforman. Al obligar a la no participación de algunas de las neuronas durante algunas etapas del entrenamiento, se evita que toda la carga de la clasificación se concentre en una sola parte de la red, haciendo que todas tengan cierta importancia a la hora de clasificar las imágnes. Según se comenta en [7], que es el paper en el que se detalla la técnica, se consigue mejorar la capacidad de predicción de redes neuronales complejas hasta en un 3 %, lo que supone una mejora notable si nos estamos moviendo en capacidades de predicción cercanas al 90 %.

También hemos introducido una modificación a la operación de entrenamiento de la red. Esta modificación es lo que se conoce como *early stopping*. La mejora consiste en establecer una condición de parada al entrenamiento, de forma que cuando parezca que el aprendizaje se ha estabilizado, se deje de entrenar la red neuronal. Esto se hace porque se considera que, una vez estabilizada la función de error durante muchas iteraciones, se ha llegado a un óptimo, probablemente no global, pero sí local. Seguir realizando iteraciones de aprendizaje una vez se ha llegado a este estado, suele tener

consecuencias negativas. Por un lado, si se sigue intentando reducir la función de error más de lo que ya se ha conseguido, se puede producir un sobreaprendizaje, que es un comportamiento bastante común en las redes neuronales. Por otro, el sobreentrenamiento de la red puede provocar que se den pasos hacia atrás, como hemos podido observar en algunos de los modelos mostrados en la práctica.

Para tratar de evitar este fenómeno, se implementa la siguiente técnica. Se va llevando un recuento de las iteraciones que han pasado desde el mejor valor de la función de pérdida obtenido hasta el momento. Si se llegan a producir un número determinado de iteraciones desde el momento en que se alcanzó el mínimo sin que se haya reducido el valor de la función, se considera que se ha ancanzado la tasa de aprendizaje óptima y se para el proceso de aprendizaje.

Para la implementación de esta técnica, utilizaremos una de las funcionalidades aportadas por TensorFlow, la cual se conoce como *Hooks*. Estos *Hooks* son fragmentos de código encapsulados en clases, que implementan ciertos métodos que se ejecutan antes y después del proceso de aprendizaje. De esta forma, en dicha clase tendremos cuatro valores, dos de ellos que se irán modificando durante la ejecución, que son el número de iteraciones que llevamos sin mejorar la función de pérdida, y el valor más bajo obtenido hasta el momento, y dos fijos que se inicializarán por el usuario al principio de la ejecución. Dichos valores son el número máximo de iteraciones que han de pasar hasta considerar que se ha estancado el aprendizaje, y el valor que límite a partir del cual se considera que se ha producido mejora.

Se ejecuta el código de esta clase cada vez que se realiza una etapa de entrenamiento. Si la función de pérdida no se ha decrementado lo suficiente (valor marcado por el límite) en el número indicado de pasos, se corta la ejecución. No se establece el límite en 0, que significaría simplemente mejorar el valor de la función de pérdida, porque una mejora muy pequeña reiniciaría el proceso, y tratamos de buscar mejoras significativas en la función de pérdida. Por este motivo se establece el umbral. En caso de que sí que se consiguiera mejorar la función de pérdida, simplemente se reinicia el contador, se actualiza el mejor valor de la función de pérdida, y se sigue el proceso de aprendizaje normalmente.

Veamos a continuación cómo se ha comportado nuestra red neuronal ante estas modificaciones.

2.6.1. Gráficas de resumen del aprendizaje

A pesar de que se han realizado varios intentos para ajustar el parámetro que mide la probabilidad de desactivar una neurona o mantenerla activa, en este documento sólo se muestra la mejor solución obtenida. El ajuste preciso de este parámetro es realmente complejo, ya que una probabilidad muy alta de descartar neuronas provoca que el modelo no ajuste bien la capacidad de predicción, y se produzca *underfitting*. Por otro lado, si la probabilidad de mantener a las neuronas es alta, se da el fenómeno contrario, y no se observan mejoras significativas por el uso de la técnica.

En nuestro caso, se ha encontrado que una aplicación agresiva de la técnica (la probabilidad de descartar una neurona era del 80 %), ha dado los mejores resultados de todas las ejecuciones. Como veremos a continuación, es el modelo que más se ha acercado a una capacidad de predicción del 90 %. Concretamente, se llega a observar en varias de las ejecuciones un valor de 89,5 % de tasa de acierto. Ninguno de los modelos anteriores había conseguido superar la barrera del 89 %, aunque sí

que habíamos tenido valores muy cercanos (el modelo en el que se basa éste, pero que no implementaba *dropout* consiguió un 88,9 %). A pesar de que no hemos conseguido aumentar tanto el valor como se conseguía en el paper, en el que se llega a una mejora de casi el 3 %, sí que hemos conseguido cierta mejora, lo que hace pensar que esta técnica puede arrojar resultados realmente prometedores.

Se probaron distintos valores de probabilidad de descarte de la neurona, como se ha dicho anteriormente. Se comenzó con una probabilidad de descarte baja, de un 40 % (concretamente, en tensorflow se especifica la probabilidad, entre 0 y 1, de conservar una neurona, por lo que establecimos el parámetro en 0,6). Con este valor ya se consiguió superar la barrera del 89 %, llegándose a una precisión en el conjunto de test de 89,2 %. Dado que la mejora nos parecía poca, se decide decrementar el valor del parámetro, hasta 0,4, es decir, una probabilidad de descarte del 60 %. Al obtener unos resultados similares, se vuelve a probar su decremento hasta 0,2, donde se consiguen los resultados mostrados. Al ver que se obtenía mejoría, se vuelve a decrementar el mismo hasta 0,1, donde nos encontramos el problema del *underfitting* que hemos comentado anteriormente. A pesar de que se consigue decrementar enormemente el valor de la función de pérdida, la capacidad de predicción de la red en test no supera el 70 %, uno de los peores resultados obtenidos en la práctica. Se decide entonces fijar el valor de probabilidad de conservar una neurona en 0,2 y terminar con la búsqueda en este contexto.
Además, en la gráfica de evolución de la función de pérdida que se adjunta, se podrá observar cómo ha funcionado el *early stopping*. En esta ejecución, se considera que se ha dejado de mejorar si se ejecutan 100 iteraciones de aprendizaje y no se ha reducido el valor de la función de pérdida al menos en 0,1. Mostramos la gráfica de evolución de la función de pérdida:

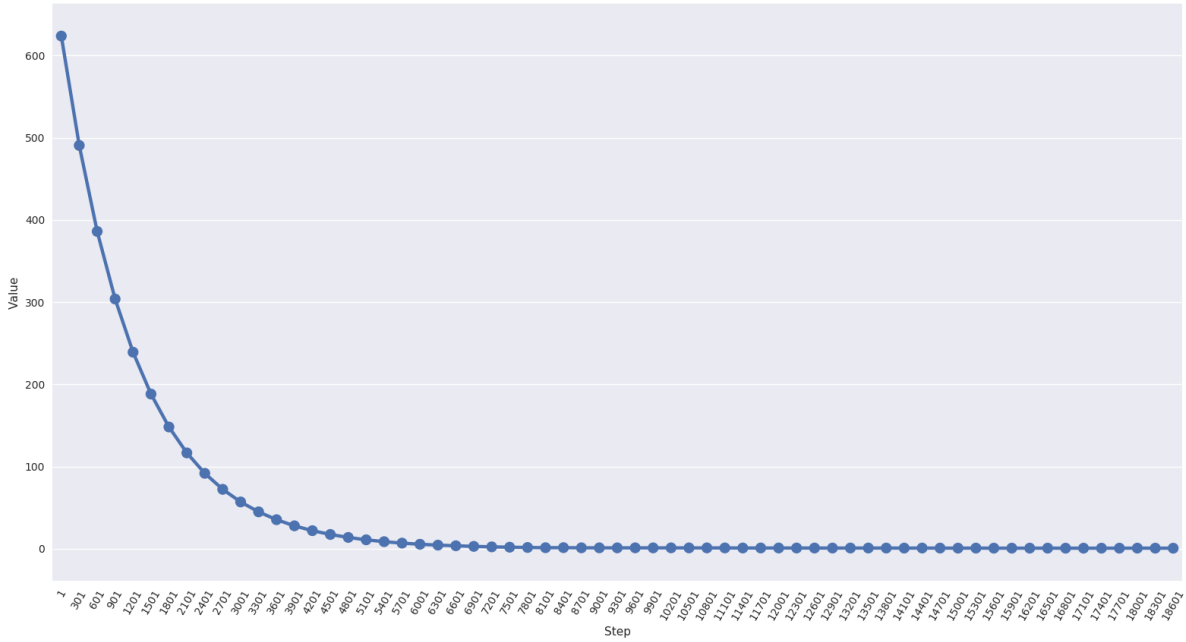


Figura 15: Evolución de la función de pérdida

Aunque sea difícilmente apreciable, dada la escala de la gráfica, cuando se llega a un estancamiento total de la función de pérdida, el modelo termina el entremiento de forma automática. Este modelo estaba programado para entrenar durante 100000 iteraciones, pero en menos de 20000 se estanca

la función de pérdida y se detiene la ejecución. En cuanto a la gráfica de capacidad de predicción, los resultados obtenidos son los que siguen:

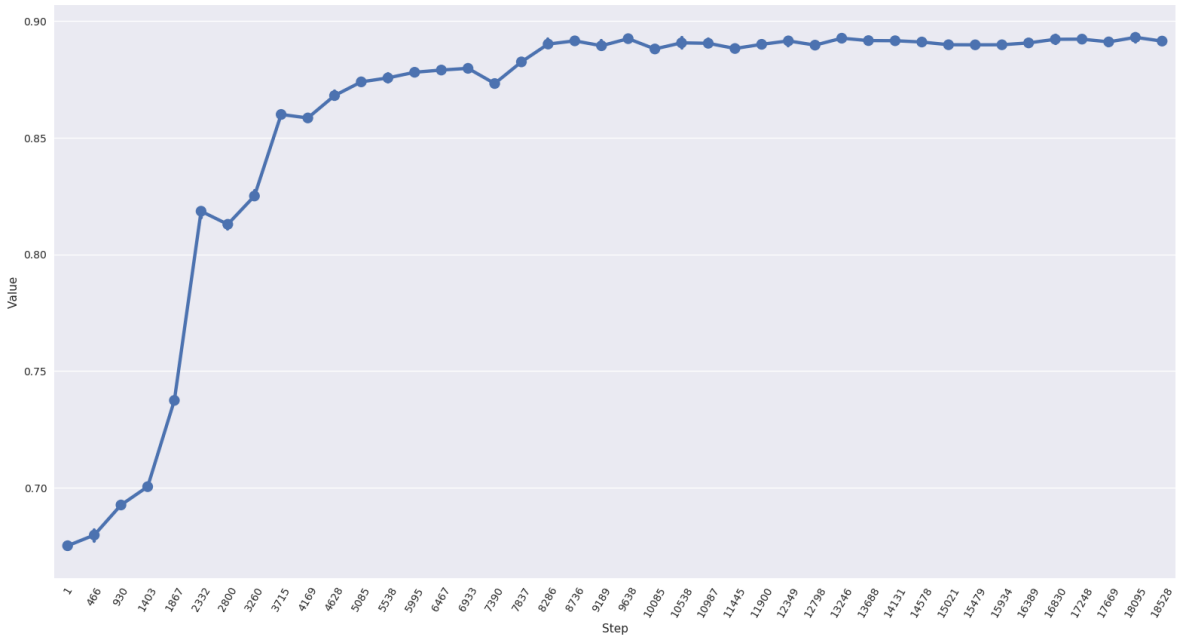


Figura 16: Evolución de la capacidad de predicción

Donde se puede observar que casi se llega al 90 %. Como ya hemos mencionado, se consigue una precisión de 89,5, que es el valor más alto obtenido hasta este punto de la práctica. De hecho, comparando las gráficas de precisión en test de los dos mejores modelos implementados, podemos observar las diferencias que hay entre ellos.

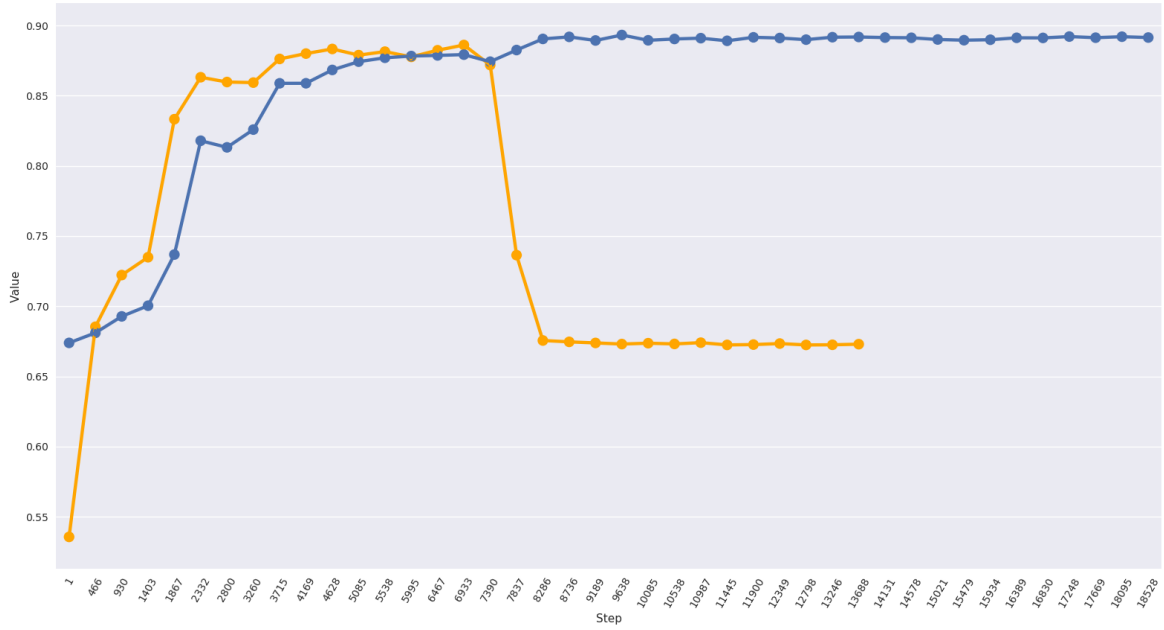


Figura 17: Comparativa de la capacidad de predicción de los dos mejores modelos

Se observa que el *dropout*, además de mejorar los resultados, parece arrojar un modelo más estable, ya que se evita el decrecimiento en la capacidad de predicción que tenía el anterior modelo. Hemos dado por tanto un paso importante en la mejora de la red neuronal convolucional.

2.7. Modelo con dos capas de convolución (2conv_model)

En este punto de la práctica, probamos a modificar la topología de la red. Hasta ahora, todos los cambios que se habían hecho consistían en introducir modificaciones y adaptaciones sobre una red neuronal con una estructura fijada. Ahora, vamos a investigar si un cambio en la estructura de la misma puede ser determinante a la hora de mejorar los resultados del aprendizaje. Es por esto que tratamos de crear una red más compleja, ahora con dos capas de convolución.

2.7.1. Estructura del modelo

La idea de crear más capas de convolución en la red viene motivada por la forma en la que se cree que el cerebro humano procesa las imágenes. De esta manera, en las capas más superficiales del cerebro, al parecer, se procesan las formas más simples, es decir, se extraen las formas más sencillas de la información recopilada por el ojo (líneas, puntos...). Esta información se transmite hacia capas más profundas de la corteza cerebral, donde se procesan estas formas simples y se agrupan para crear elementos más complejos (curvas, polígonos, intersecciones...). De esta forma, se parte de un pequeño conjunto de formas procesadas por unas pocas neuronas, hasta figuras de alta complejidad, como puede ser una persona. Usualmente, el conjunto de formas simples es más o menos reducido, mientras que conforme va aumentando la complejidad de las mismas, debido a que las formas simples se pueden combinar de multitud de maneras distintas, el número de formas complejas distintas aumenta.

Este argumento es el que sustenta la estructura que se ha dado a esta red neuronal. Comenzaremos con una capa de convolución más o menos básica, que sólo extrae 40 filtros. A continuación, tendremos otra capa de convolución, que extraerá 80 filtros nuevos de los 40 filtros anteriores. Después, tendremos la capa de alisamiento, para convertir la información almacenada de forma matricial en vector, y dos capas completamente conectadas antes de la capa de salida. La primera de ellas tendrá 200 neuronas, y la segunda 100. La idea de tener dos capas completamente conectadas en lugar de una sigue la misma lógica que teníamos anteriormente, aunque ahora de forma inversa. Vamos a ir agrupando la información extraída en capas de menor tamaño, pero cuyas neuronas vayan extrayendo información del agrupamiento de formas complejas. Como el problema que nos ocupa consiste simplemente en diferenciar animales de dos clases, no necesitamos un conjunto de neuronas muy grande para las capas de salida, si no que puede ser interesante tener pocas neuronas que realicen tareas bastante específicas.

Se adjunta a continuación el esquema de la red neuronal implementada:

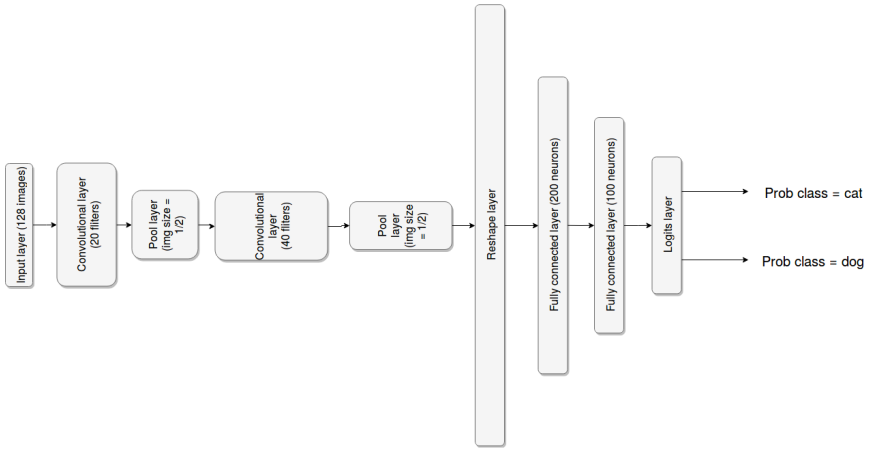


Figura 18: Esquema del modelo de red neuronal convolucional con dos capas

En cuanto al esquema de aprendizaje, hemos visto anteriormente que los modelos que combinaban un *Weight Decay* con un parámetro bajo con una tasa de aprendizaje variable utilizada en el método de gradiente descendente tenían un funcionamiento bastante apropiado, así que para la primera prueba utilizaremos este esquema de aprendizaje.

2.7.2. Gráficas de resultados obtenidos

Al igual que con los modelos anteriores, vamos a mostrar las gráficas que muestran el desarrollo del proceso de aprendizaje de la red neuronal:

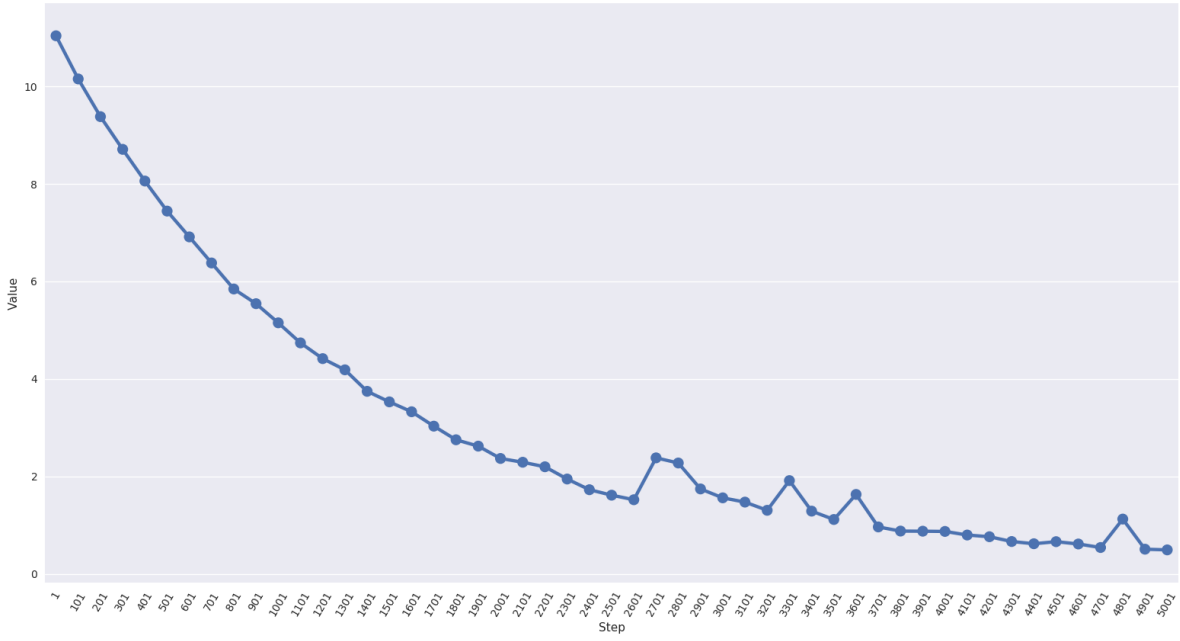


Figura 19: Evolución de la función de pérdida

Como podemos observar, hemos conseguido llegar a un valor de la función de pérdida muy cercano a 0. Esto significa que esta arquitectura consigue optimizar bastante bien la función de pérdida, y

en un número de pasos relativamente rápido (sólo se han ejecutado 5000 pasos de aprendizaje). No obstante, en la gráfica siguiente podemos observar que, a pesar de que se ha conseguido reducir la función de pérdida mucho, el porcentaje de acierto conseguido por la red no es tan bueno como se podría esperar:

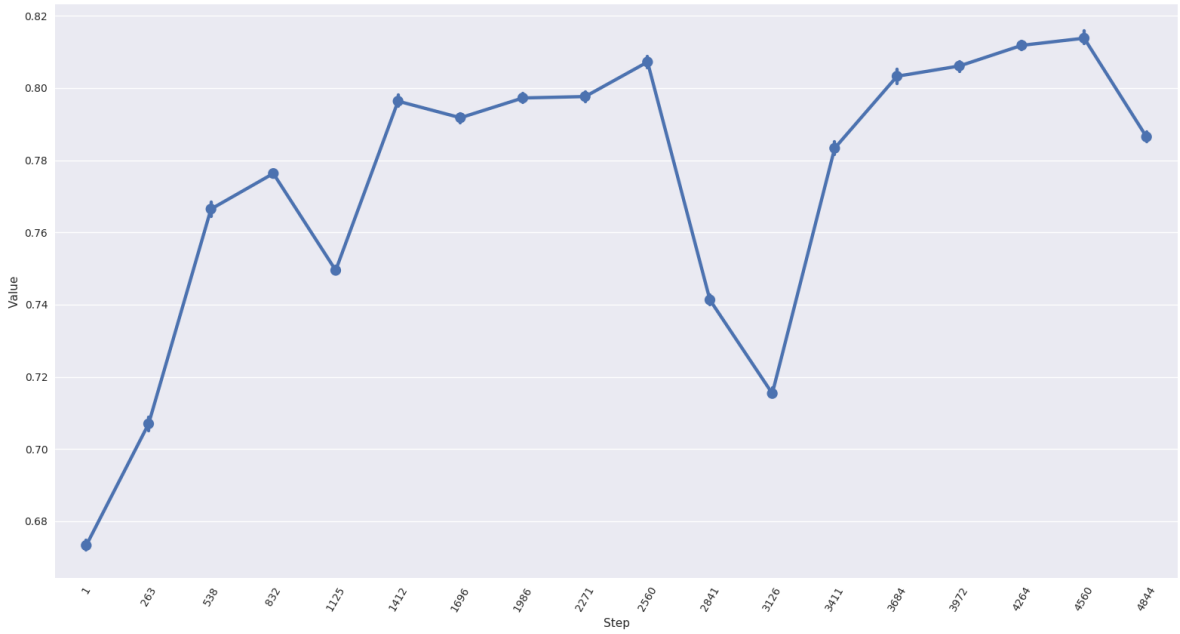


Figura 20: Evolución de la capacidad de predicción

Podemos observar, primeramente, que durante la ejecución se produjo un importante paso hacia atrás en la capacidad de aprendizaje, en torno al paso 3000. No es sencillo explicar este comportamiento, aunque teniendo en cuenta el crecimiento de la función de pérdida en torno al paso 2700, es posible que el lote de imágenes que se generase fuera especialmente complicado de tratar, y se produjera una explosión en el gradiente descendente, haciendo que la red perdiese capacidad de aprendizaje. En cualquier caso, a pesar de que la red consigue resultados bastante buenos, llegando hasta una capacidad de predicción de un 82 % en apenas 4500 pasos de aprendizaje, no son del todo satisfactorios. Hemos escogido una red de mayor complejidad que las anteriores, y a pesar de ello los resultados son peores que los obtenidos por redes más sencillas.

Esto, unido a las restricciones de hardware que tenemos (usualmente, las redes neuronales se entrenan en ordenadores de mucha capacidad, con unidades de cálculo optimizadas para el trabajo vectorial), nos hace abandonar esta línea, ya que la modificación de parámetros y reentrenamiento de las redes es un proceso muy costoso y que consume mucho tiempo. Como no encontramos indicios de que esta línea pudiera llevar a resultados realmente interesantes, se decide que no merece la pena la inversión de tiempo en este sentido, e invertir ese tiempo en otros de los modelos que aparecen comentados a lo largo de la práctica.

2.8. 3 capas de convolución: aprendizaje mediante adamOptimizer (3adam_model)

A lo largo de toda la práctica hemos trabajado con el mismo esquema de aprendizaje: *Gradient Descent*. En este apartado trataremos con uno diferente, *AdamOptimizer*. *AdamOptimizer* consiste

en un método estocástico para determinar los pesos de nuestra red neuronal. *AdamOptimizer* trata de optimizar el método de gradiente descendente siendo más eficiente temporalmente, así como conseguir una mejor variación de la tasa de aprendizaje que se ajuste a nuestro problema combinando para ello los métodos de optimización de los momentos y *RMSProp*.

Aunque generalmente arroja mejores resultados que el gradiente descendente, en general necesita de redes neuronales con topologías más complejas. Trataremos en esta ocasión con una red neuronal de 3 capas de convolución y pooling, ya que contamos con restricciones hardware, como se ha comentado en el apartado anterior. A pesar de ello, se implementa esta red para tratar de probar otros esquemas de aprendizaje, ya que en la literatura especializada muchos autores coinciden en que el esquema de aprendizaje *Gradient Descent* es muy arcaico y los resultados obtenidos por el mismo son muy mejorables. En la sección siguiente definimos en mayor profundidad la arquitectura elegida.

2.8.1. Estructura de la red

Como ya se ha comentado, en esta ocasión contaremos con 3 capas de convolución y pool, manteniendo igual el resto de la estructura que presentabamos en el primer modelo. La primera capa de convolución recibirá un batch de 128 imágenes de tamaño $64 \times 64 \times 3$ y extraerá un total de 64 filtros para cada una de las imágenes. Seguido de esto tendremos una capa de pool que reduzca el tamaño de las imágenes a la mitad, produciendo así una salida de tamaño $128 \times 32 \times 32 \times 64$. A continuación contaremos con una capa de convolución y una capa de pool igual a las anteriores. Por tanto, en este punto contaremos con un conjunto de imágenes de tamaño $128 \times 16 \times 16 \times 64$. Finalmente, se tiene una capa de convolución que extraerá 128 filtros para cada foto, seguida de una capa de pool que de nuevo reducirá el tamaño de las imágenes a la mitad, quedándonos así un conjunto de tamaño $128 \times 8 \times 8 \times 128$. Esta salida será tomada como entrada para una capa de redimensión, que nos producirá una salida de 128×8192. El resto es idéntico al expuesto en el modelo básico.

En la siguiente imagen podemos ver el esquema descrito:

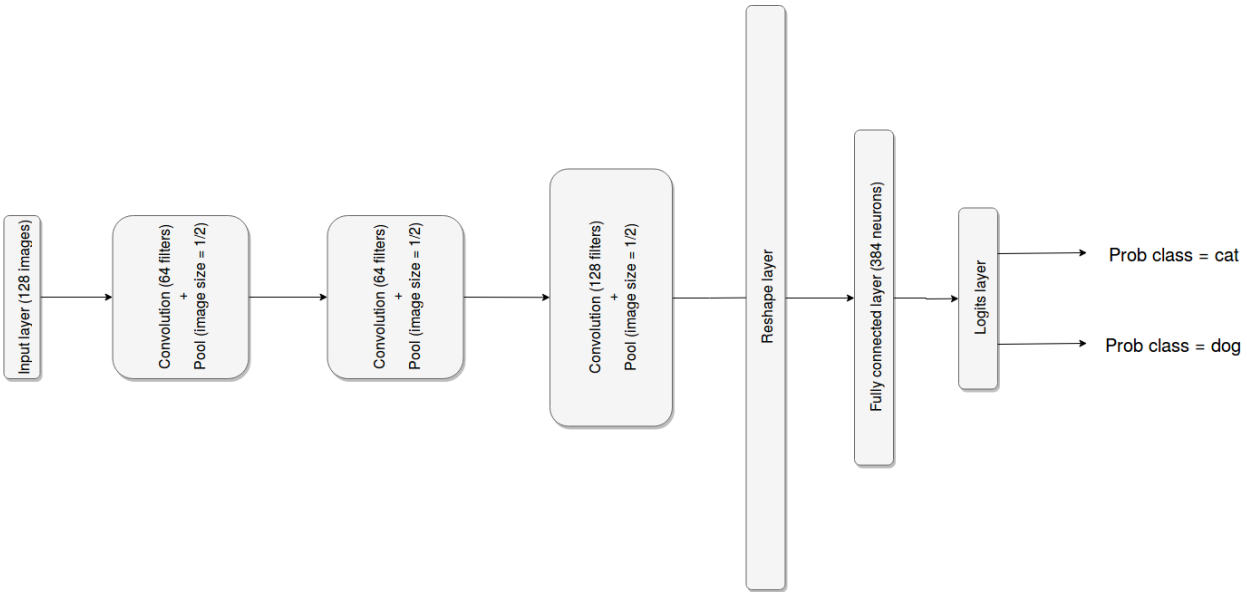


Figura 21: Esquema del modelo de red neuronal convolucional

2.8.2. Gráficas de resumen del aprendizaje

Una vez explicada la arquitectura de la red, pasamos analizar el comportamiento de la misma.

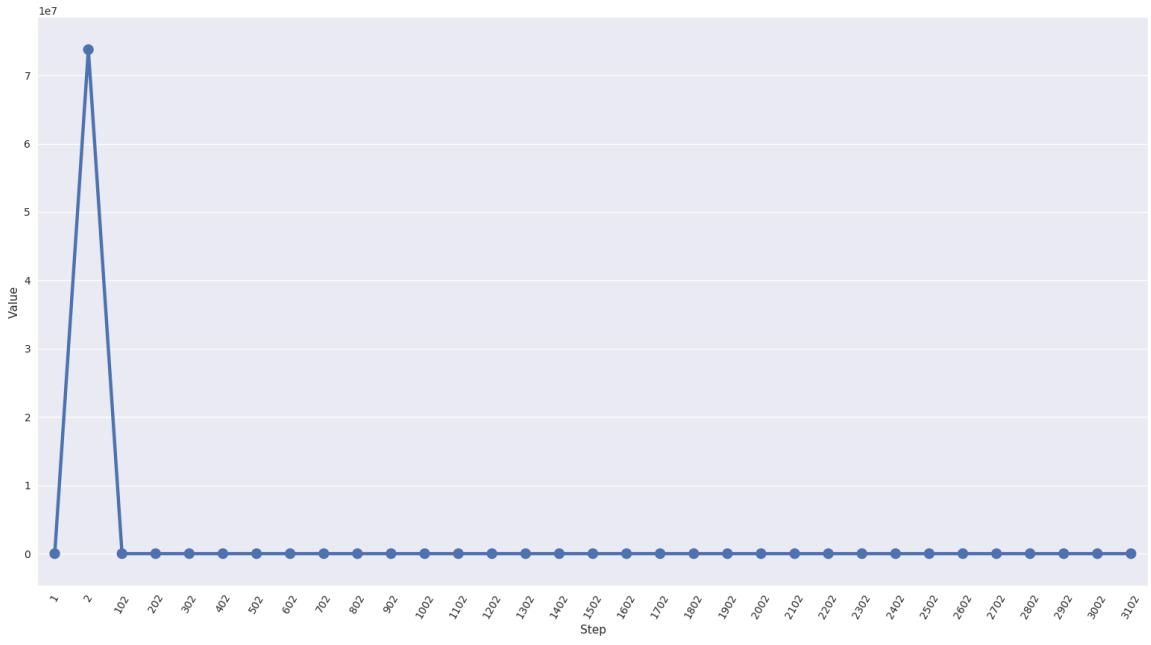


Figura 22: Evolución de la función de pérdida

En la gráfica anterior tan solo podemos ver que, en el momento inicial, la función de pérdida alcanza un valor del orden de 10^7. Sin embargo, esto tan solo es consecuencia de la inicialización, ya que encontramos que inmediatamente el valor tomado por la función de pérdida desciende. Poco más se puede extraer de la gráfica anterior, ya que los datos a partir del paso 100 son prácticamente

nulos en comparación con el inicial. Para poder estudiar mejor el comportamiento de esta función vamos entonces a mostrar la gráfica de la función de pérdida a partir del paso 200.

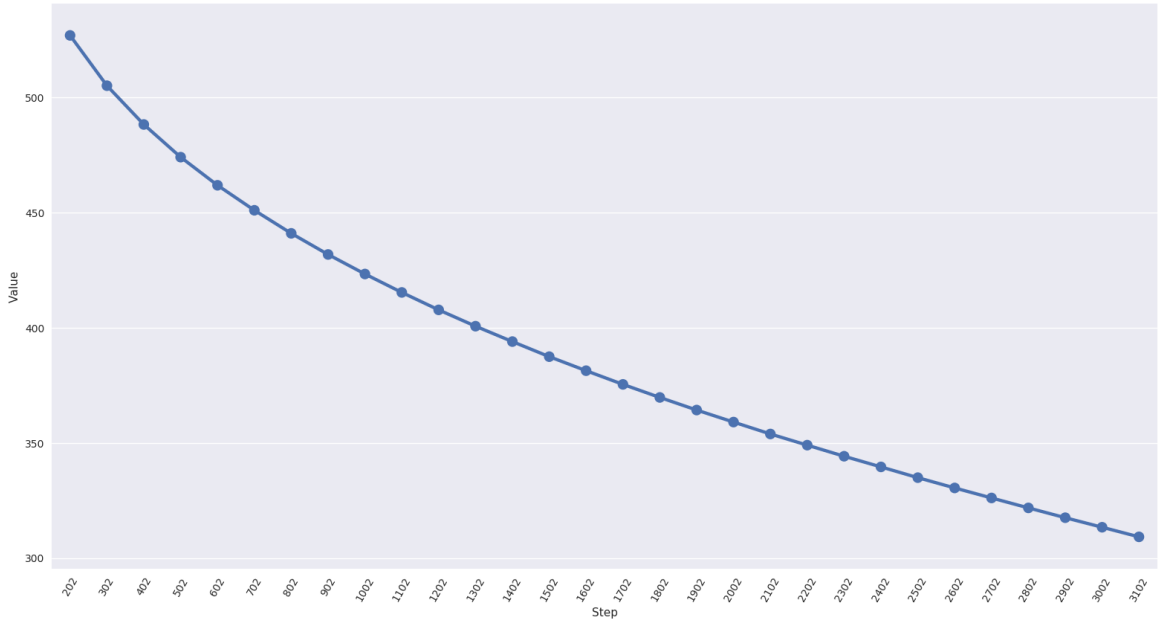


Figura 23: Evolución de la función de pérdida

Ahora sí podemos observar como la función de pérdida desciende de forma regular hasta alcanzar un valor de casi 300. A pesar de que los valores siguen siendo considerablemente altos y la pendiente de la curva anterior considerable, se paró el entrenamiento tras 8 horas desde su inicio.

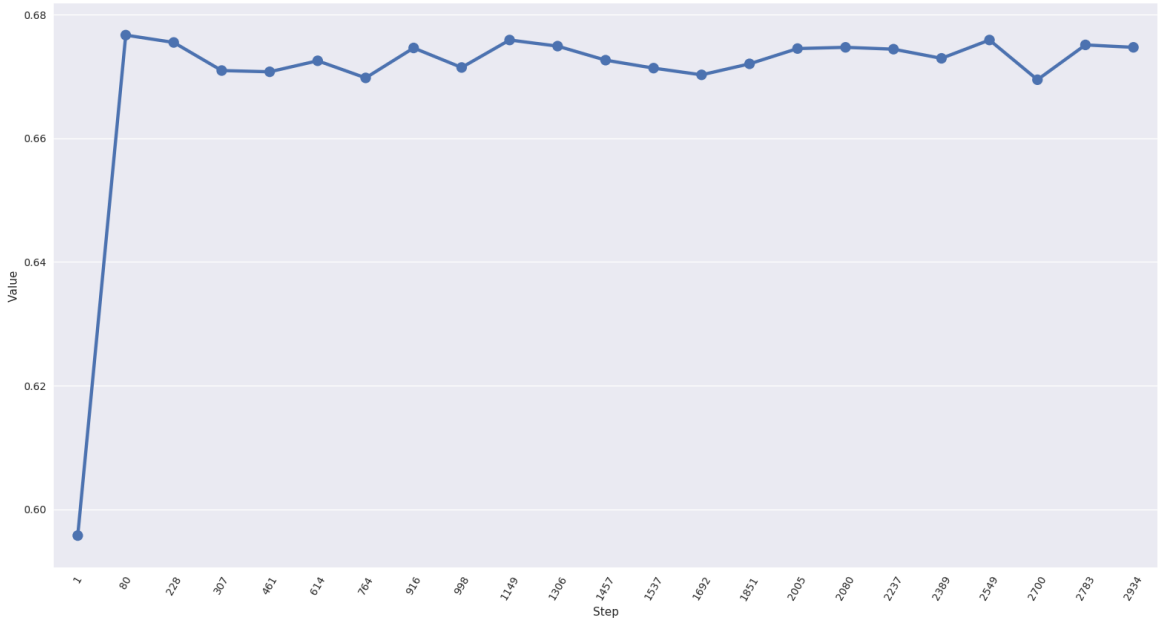


Figura 24: Evolución de la capacidad de predicción

Sobre la gráfica anterior, nos muestra una capacidad de predicción más o menos estable, con diferencias de como mucho un 1 % a lo largo de 3000 pasos, lo cual no es muy prometedor. En cuanto a la capacidad de predicción en sí, en ningún punto llega a alcanzar el 68 % de acierto. Estos suponen los peores resultados obtenidos hasta el momento. Además, ha sido la red que más recursos temporales ha consumido, por lo que este modelo quedaría descartado.

Probablemente con mayor capacidad de cómputo, y por tanto la posibilidad de crear una red más compleja, con un mayor número de capas, y cada una con más neuronas, conseguiríamos resultados considerablemente mejores gracias a *AdamOptimizer*.

3. Conclusiones

A lo largo de la práctica se han implementado modelos de redes neuronales convolucionales para estudiar cómo se comportan los mismos a la hora de la tarea de la clasificación de imágenes, afrontanto para ello el problema de distinguir imágenes de gatos y perros.

Con los distintos modelos que se han implementado, se ha podido observar que, aunque un esquema simple de red neuronal convolucional puede no aportar resultados muy satisfactorios, la introducción de modificaciones puede conseguir que un modelo que a priori es de mala calidad consiga beneficios notables. Esto se pone de manifiesto muy claramente cuando se incorporan métodos de regularización como *Weight Decay*, o se utilizan tasas de aprendizaje variables. No obstante, también hemos podido comprobar que el ajuste de parámetros de las modificaciones puede no ser sencillo, y que métodos que independientemente deberían aportar mejoras, cuando se combinan pueden dar lugar a resultados no satisfactorios. Es necesario estudiar la estructura detalladamente para tratar de conseguir los resultados óptimos.

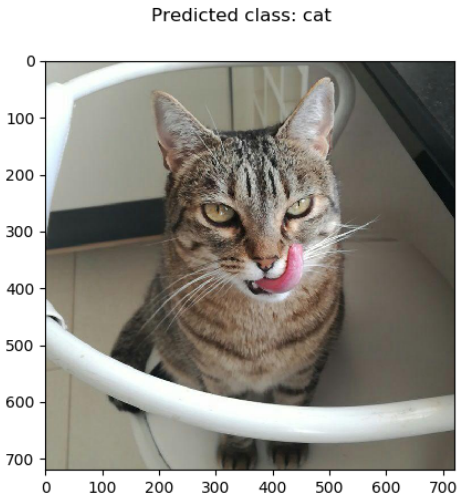
Se ha puesto de manifiesto también que las necesidades de cómputo a la hora de entrenar redes neuronales convolucionales es alta. Se ha estado trabajando con redes con una única capa de convolución, que ha dado resultados bastante buenos, pero probablemente se podrían haber mejorados estos con arquitecturas más profundas. Las limitaciones impuestas por el uso de los ordenadores personales, y los malos resultados obtenidos en las primeras ejecuciones en las que se abordaban redes más profundas nos hizo abandonar rápidamente esta línea de investigación.

Otro punto que puede ser destacable y que limita la capacidad de aprendizaje de la red neuronal es el conjunto de datos utilizado. Tal y como se describió al principio, nos encontramos ante un conjunto de datos bastante desbalanceado, ya que se dispone del doble de imágenes de perros que de gatos. A pesar de que no parece haber tenido mucha afectación en los resultados obtenidos, es posible que un mayor balanceo en las clases hubiera aportado mejores resultados.

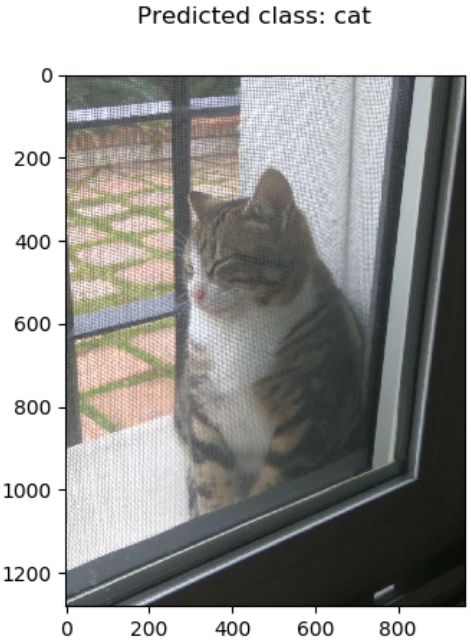
3.1. Uso final de la red neuronal para la predicción de imágenes

Una vez concluida la etapa de investigación, se trata de poner en aplicación real el modelo entrenado. Para ello, se ha desarrollado un programa simple, que lee una imagen pasada como argumento, y trata de clasificar la misma utilizando el mejor modelo de red neuronal que se ha obtenido. Una

vez clasificada la imagen, se muestra por pantalla la misma, junto con la predicción realizada. A continuación se muestran las ejecuciones de dicho programa para distintas imágenes:

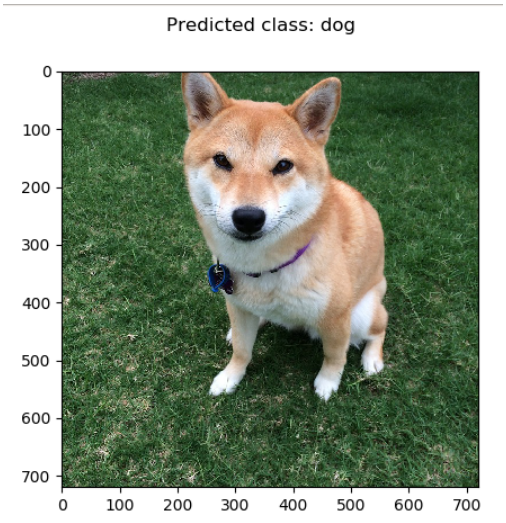


(a) Predicción sobre el gato de María del Mar

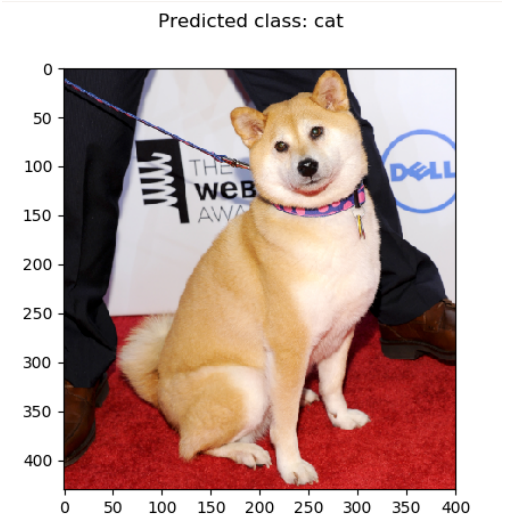


(b) Predicción sobre el gato de Paco

Como se puede observar, la red neuronal tiene capacidad para reconocer gatos en las imágenes, y para las dos imágenes de prueba que se han aportado, consigue hacer una predicción correcta. No obstante, como veremos a continuación, la red entrenada no es perfecta, y puede cometer errores de predicción:



(a) Predicción correcta de un perro



(b) Predicción incorrecta de un perro (cree que es un gato)

3.2. Ejecución de la práctica para contrastar los resultados

Para evitar adjuntar toda la información de todos los modelos entrenados, ya que se han generado más de 4GB de información entre todos ellos, pero que esta última parte de la práctica sí que pueda ser ejecutada, se ha subido a Consigna UGR una carpeta con el modelo más potente ya entrenado. En dicha carpeta se puede encontrar el modelo entrenado, las imágenes que se han usado para el entrenamiento y el test, y las imágenes que se han utilizado como ejemplo para comprobar el buen funcionamiento de la red con ejemplos de imágenes reales. Además, aparecen los archivos de texto que especifican el conjunto de entrenamiento y test. Para ejecutar el modelo, hay que dirigirse a la carpeta `submit_model` y ejecutar el archivo `predict_images.py`. Una vez ejecutado, se mostrarán por pantalla, de forma ordenada, las cuatro imágenes predichas. Es necesario tener instalado TensorFLow en su versión más reciente (1.4) para poder ejecutarlo. También puede reentrenarse este modelo ejecutando el archivo `model_train.py`, o evaluar su capacidad de predicción sobre el conjunto de test ejecutando el archivo `model_test.py`. La URL desde la que se pueden descargar estos archivos es https://consigna.ugr.es/f/3pN90VdApiS8OEi0/submit_model.zip

Referencias

[1] Oxford University. *The Oxford-IIIT Pet Dataset*. URL: http://www.robots.ox.ac.uk/~vgg/data/pets/.

[2] Weiwei Zhang et al. *Cat Dataset*. URL: https://archive.org/details/CAT_DATASET.

[3] Stanford University. *Stanford Dogs Dataset*. URL: http://vision.stanford.edu/aditya86/ImageNetDogs/.

[4] Google Inc. *TensorFlow*. URL: https://www.tensorflow.org.

[5] Google Inc. *TensorFlow - Convolutional Neural Networks*. URL: https://www.tensorflow.org/tutorials/deep_cnn.

[6] Aurélien Géron. *Hands on Machine Learning with Scikit-Learn & Tensorflow*. 1.ª ed. O'Reilly, 2017. ISBN: 1491962291.

[7] Nitish Srivastava y col. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". En: *J. Mach. Learn. Res.* 15.1 (ene. de 2014), págs. 1929-1958. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=2627435.2670313.