Universidad de Granada

Visión por computación

Clasificación de imágenes usando CNNs

Francisco Luque Sánchez
María del Mar Ruiz Martín

13 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:

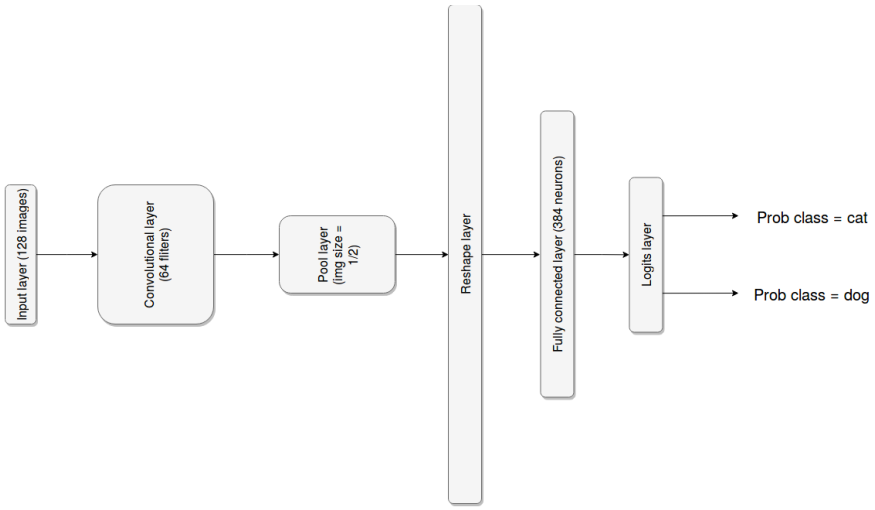

Figura 1: Esquema del modelo simple de red neuronal convolucional

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 nueronal. 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:

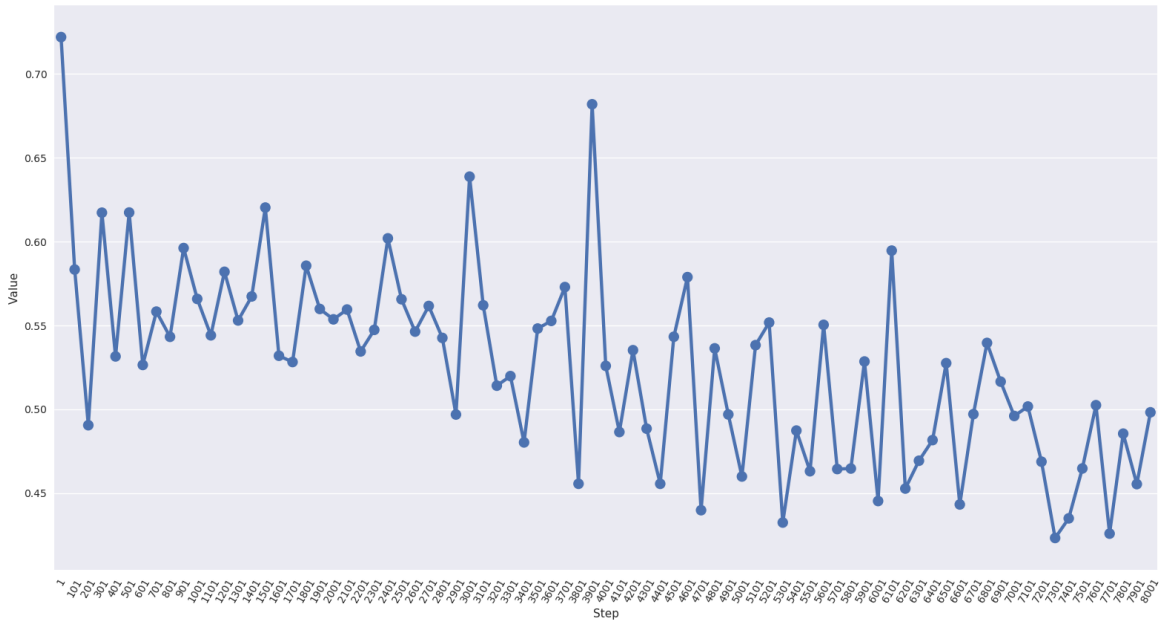

Figura 2: Evolución de la función de pérdida

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:

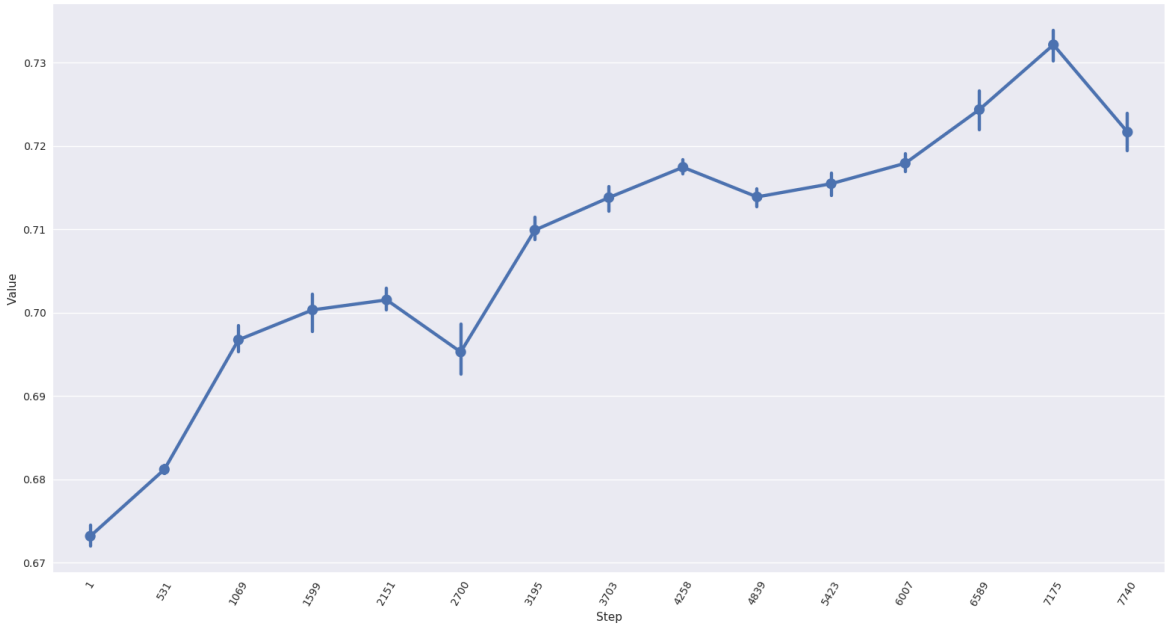

Figura 3: Evolución de la precisión de la red neuronal

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* simple sobre los pesos de la red neuronal. Pasamos a describirlo y ver los resultados en la siguiente sección.

2.2. Primera mejora: *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 si 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

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.